UNIVERSIDAD TECNOLOGICA NACIONAL
DEPARTAMENTO DE SISTEMAS DE INFORMACION
CATEDRA :
PARADIGMAS DE PROGRAMACION

APUNTE TEORICO-PRACTICO

UNIDAD 1:
EL PARADIGMA ORIENTADO A OBJETOS

AUTORES Y COLABORADORES

TOPICOS INTRODUCTORIOS

Ing. Karina Ligorria
Analista U. de Sistemas Javier Ferreyra

UNIDAD 1

Ing. Analía Guzmán
Ing. Karina Ligorria
Analista U. de Sistemas Javier Ferreyra
Ing. Gustavo Garcia
Ayudante alumno Guillermo Colazo
Ayudante alumno Soledad Alborno

COMPAGINACION

Ing. Analía Guzmán

AÑO 2004

INDICE GENERAL

TOPICOS INTRODUCTORIOS I

UNIDAD Nº 1 – EL PARADIGMA ORIENTADO A OBJETOS
PARTE TEORICA

CONCEPTOS GENERALES DE LA PROGRAMACION ORIENTADA A OBJETOS
1. INTRODUCCION 1
2. COMPONENTES BASICOS 2
   - OBJETO 2
   - MENSAJES 3
3. CARACTERISTICAS DE LA PROGRAMACION ORIENTADA A OBJETOS 3
4. VENTAJAS DE LA PROGRAMACION ORIENTADA A OBJETOS 6
5. DESVENTAJAS DE LA PROGRAMACION ORIENTADA A OBJETOS 7
6. LENGUAJES DE LA PROGRAMACION ORIENTADA A OBJETOS 8

PROGRAMACION ORIENTADA A OBJETOS EN JAVA
1. INTRODUCCION 10
2. CLASES 10
   - INTRODUCCION 10
   - ESTRUCTURA GENERAL 11
   - DECLARACION Y DEFINICION 12
   - CUERPO DE UNA CLASE 12
   - INSTANCIAS DE UNA CLASE (OBJETOS) 12
     - Acceso a los miembros 13
     - La vida de un objeto 13
     - Arrays de objetos 14
   - DECLARACION DE MIEMBROS DE CLASES 15
     - Modificadores de acceso a miembros de clases 16
   - ATRIBUTOS DE UNA CLASE 20
     - Miembros estáticos 21
   - METODOS DE UNA CLASE 23
     - Declaración y definición 23
     - El cuerpo 24
       - Declaración de variables locales 24
       - Llamadas a métodos 24
         - El objeto actual (this) 25
     - Métodos especiales 26
       - Sobrecargados 26
         - Resolución de llamadas a métodos 26
       - Constructores 27
         - Tipos de constructores 29
         - Caso especial 30
3. HERENCIA 31
   - INTRODUCCION 31
     - Heurísticas para crear subclases 31
   - HERENCIA SIMPLE 32
     - Clases bases 33
     - Clases derivadas 33
       - Declaración 33
       - Atributos y métodos 34
       - Acceso a miembros de clases bases 34
       - Constructores en clases derivadas 37
   - REDEFINICION DE METODOS 38
     - Redefinición y sobrecarga 39
     - Caso especial (especificador final) 39
   - JERARQUIA DE CLASES 40
     - Un ejemplo de jerarquía de clases 40
       - La clase base 40
       - Objetos de la clase base 41
       - La clase derivada 41
       - Objetos de la clase derivada 43
4. POLIMORFISMO 43
   - INTRODUCCION 43

METODOS Y CLASES ABSTRACTAS 43
Interface 45
Diferencias entre interfaces y clases abstractas 46
ENLACE DINAMICO 46
IMPLEMENTACION 47
Implementación a través de un ejemplo 48
La jerarquía de clases que describen la figuras planas 48
La clase Figura 48
La clase Rectangulo 49
La clase Circulo 49
Uso de la jerarquía de clases 50
Enlace dinámico 50
El polimorfismo en acción 50
Añadiendo nuevas clases a la jerarquía 52
La clase Cuadrado 52
La clase Triangulo 52
El polimorfismo en acción 52

PARTE PRACTICA

INTRODUCCION AL LENGUAJE JAVA

1. INTRODUCCION 55
CARACTERISTICAS 55
2. INSTALACION DE JAVA 56
3. COMPILACION Y EJECUCION DE UN PROGRAMA 56
4. GRAMATICA
COMENTARIOS 59
IDENTIFICADORES 60
TIPOS DE DATOS 60
VARIABLES 61
Genero 62
Asignación 62
Inicialización 62
OPERADORES 63
SEPARADORES 64
ESTRUCTURAS DE CONTROL 65
5. UN PROGRAMA JAVA 67
6. ARREGLOS 69
7. CADENAS DE CARACTERES 70
8. REFERENCIAS EN JAVA 72
REFERENCIAS Y ARRAYS 74
REFERENCIAS Y LISTAS 74
9. PAQUETES 76
EL PAQUETE (PACKAGE) 76
LA PALABRA RESERVADA IMPORT 76
PAQUETES ESTANDAR 77
RESUMEN 77
10. MANEJO BASICO DE ERROR Y EXCEPCION 78
EXCEPCIONES EN JAVA 78
CAPTURA DE EXCEPCIONES 79
TIPOS DE EXCEPCIONES 80
UN EJEMPLO QUE MANEJA EXCEPCIONES 81
11. ENTRADAS Y SALIDAS 85
12. LIBRERIAS ESTANDAR 89
13. AYUDA EN LINEA 90

PRESENTACION DE CODIGO CON UML

1. INTRODUCCION 92
2. JAVA Y UML 92
3. TERMINOS DE JAVA Y UML 93
4. COMPONENTES DE LOS DIAGRAMAS UML 93
5. ICONOS DE ACCESIBILIDAD 94

EJERCICIOS PRACTICOS

1. EJERCICIOS RESUELTOS 96
2. EJERCICIOS PARA RESOLVER 115

TOPICOS INTRODUCTORIOS

QUÉ ES UN PARADIGMA?

Según el historiador Thomas Kuhn un paradigma es: “un conjunto de teorías, estándares y métodos que en conjunto representan una forma de organizar el conocimiento, es decir, una forma de ver la realidad”.

PARADIGMAS DE PROGRAMACION

Es un conjunto de elementos, estructuras y reglas para construir programas. Cada paradigma define un programa con diferentes términos.

Básicamente, un programa es la “simulación computacional de una porción de la realidad”. No exactamente la realidad sino lo que creemos haber comprendido de ella. Por ser este un modelo de la realidad existe una distancia con la realidad que simula: GAP[1] Semántico. Dicha distancia depende del esfuerzo que se debe realizar para reflejar la realidad.

El mejor paradigma es aquel que tiene un menor GAP semántico.

De acuerdo con el Paradigma de Programación, podemos clasificar los lenguajes en las siguientes categorías:

- **Imperativos**: Son aquellos lenguajes, que basan su funcionamiento en un conjunto de instrucciones secuenciales, las cuales, al ejecutarse, van alterando las regiones de memoria donde residen todos los valores de las variables involucradas en el problema que se plantea resolver. Es decir, se cambia progresivamente el estado del sistema, hasta alcanzar la solución del problema [CONTRERAS 01].
  Como un ejemplo ilustrativo vamos a escribir un programa en un lenguaje de este tipo para calcular el factorial de un número positivo x.

```
READ(x);
fac := 1 ;
for i = 1 to x
{
   fac := fac * i ;
}
WRITELN(fac);
```

- **Declarativos**: En este paradigma, más que el ¿cómo? desarrollar paso a paso un proceso, nos interesa el ¿qué? deseamos obtener a través del programa. Quizás el lenguaje declarativo que nos sea más familiar, es SQL, el cual es utilizado para interactuar con la información de bases de datos, concentrándose (como se podrá observar en el siguiente ejemplo), sólo en los resultados que van a ser obtenidos, dejándole al traductor la tarea de cómo llegar a ellos y presentárnoslos.[SANDERS-PRICE 02]

```
SELECT * FROM alumnos WHERE sexo = "M" ORDER BY edad
```

  Dentro de este paradigma, se encuentran dos estilos distintos de programación, cada uno de los cuales posee su propia lógica [SANFÉLIX 00].

  - **Funcionales**: Son lenguajes basados en funciones, las cuales se representan mediante expresiones, que nos permiten obtener ciertos resultados a partir de una serie de argumentos[BIBBY 00]. De hecho las expresiones están formadas por un conjunto

[1] GAP: separación

de términos, que a su vez pueden encapsular otras expresiones, para con la evaluación de todas ellas, llegar a la solución deseada.[GAULD 01]. Para describir la idea, retomaremos el ejemplo del factorial escrito en el lenguaje funcional Haskell.

```
fac :: Integer -> Integer
fac 0 = 1
fac x = x * fac (x-1)
```

- **Lógicos**: Este tipo de lenguajes se basan en el cálculo de predicados, la cual es una teoría matemática que permite entre otras cosas, lograr que un ordenador basándose en un conjunto de hechos y de reglas lógicas, pueda derivar en soluciones inteligentes. [DIMARE 90]. El mismo ejemplo del factorial, se vería de la siguiente manera, escrito en PROLOG.

```
factorial (0, 1)
factorial (X, Fac) :- Y is X-1,   fac(Y, F2),  Fac is F2 * X .
```

- **Orientados a Objetos**: Este último paradigma, como se puede observar en la figura 1, algunas veces se mezcla con alguno de los otros 2 modelos, sin embargo mantiene características propias, que lo diferencian claramente. Los programas de este tipo, se concentran en los objetos que van a manipular, y no en la lógica requerida para manipularlos [MARBUS 00]. Ejemplos de objetos pueden ser: estudiantes, coches, casas etc, cada uno de los cuales tendrá ciertas funciones (métodos) y ciertos valores que los identifican, teniendo además, la facultad de comunicarse entre ellos a través del paso de mensajes.

Tratando de resumir un poco, presentaremos los siguientes cuadros evolutivos, donde aparecen los lenguajes que por su uso y comercialización, han resultado ser los más populares a lo largo de este medio siglo. [LABRA 98] [RUS 01]

Figura a. Evolución de los Lenguajes Imperativos y Orientados a Objetos

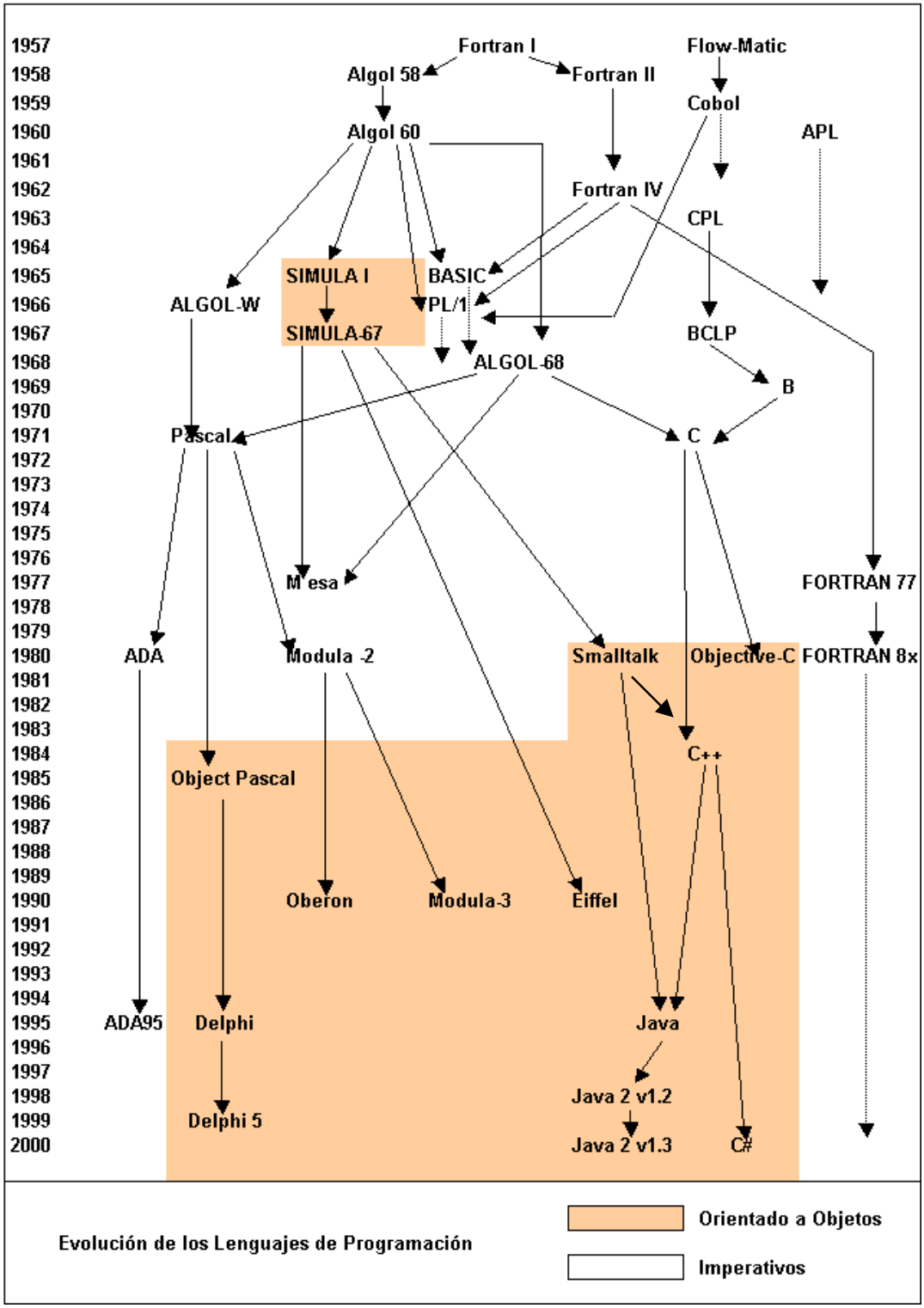
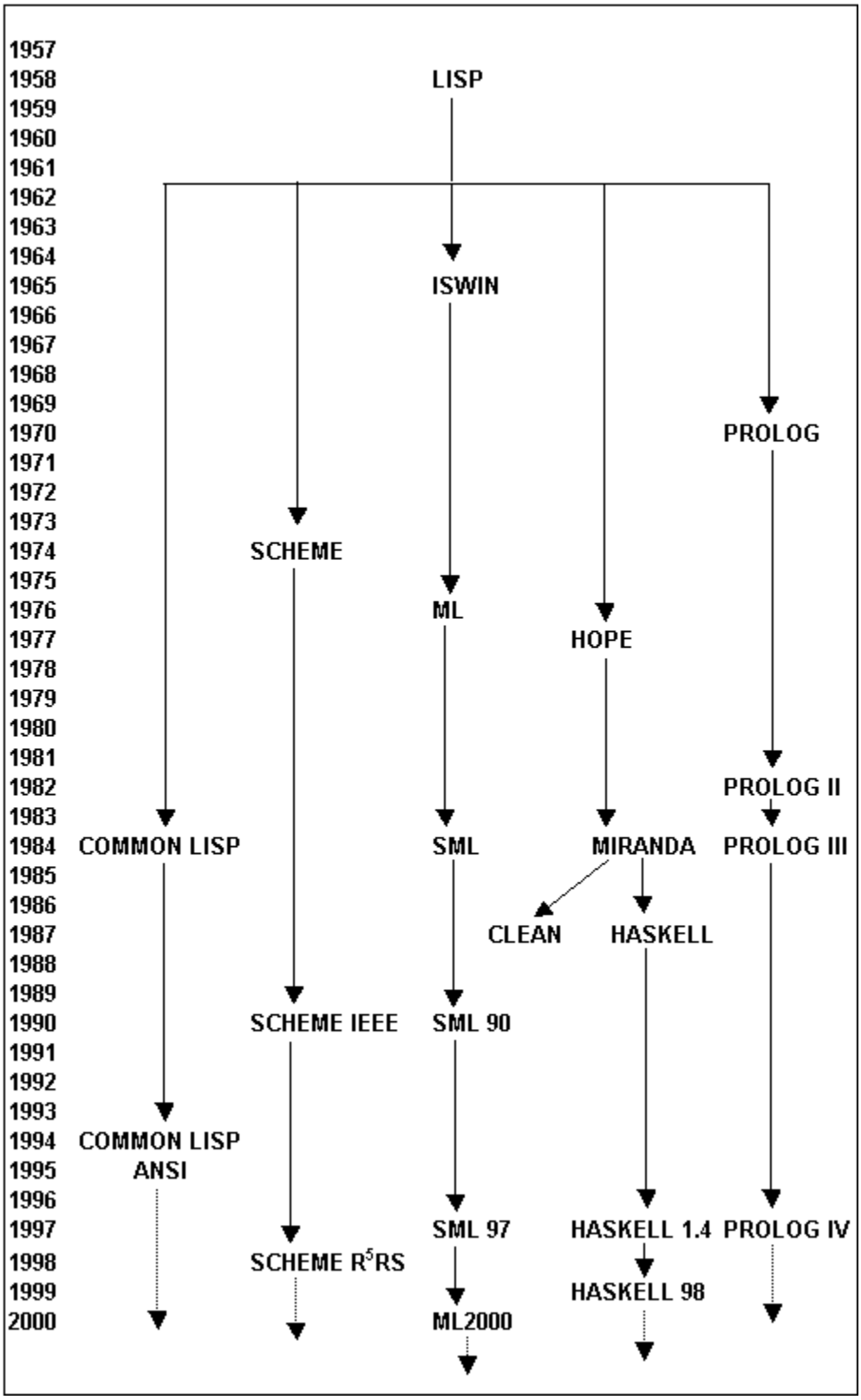


Figura b. Evolución de los lenguajes declarativos



Como ya lo citamos anteriormente y como se puede observar en las figuras a y b, la existencia de tantos lenguajes obedece a que cada uno de ellos está encaminado a resolver ciertas tareas, dentro de la amplia problemática de la explotación de la información, o bien, a que su arquitectura, o su forma de llevar a cabo la programación, tiene un enfoque particular.

Ahora bien, si tomamos como referencia las herramientas usadas en el proceso de traducción y ejecución de los programas esbozada en la figura 2, vamos a tener la siguiente clasificación de lenguajes[AHO 77]:

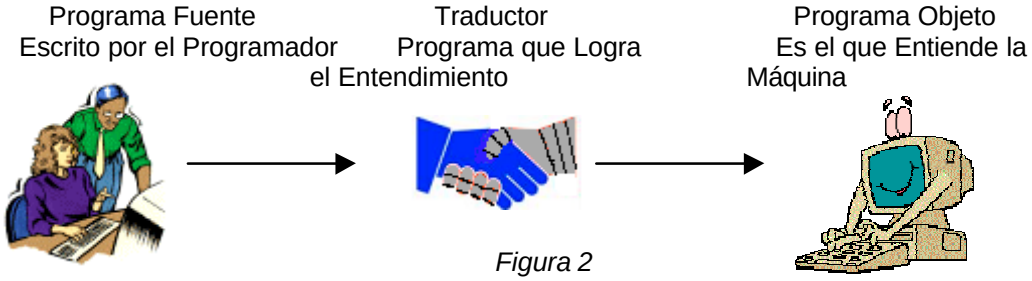


Figura 2

- **Lenguajes Ensamblados**: Se refieren al lenguaje ensamblador, que viene a ser una representación simbólica de las instrucciones correspondientes al lenguaje ensamblador de alguna arquitectura específica, con lo que, casi siempre, la correspondencia entre las instrucciones de este lenguaje, y las del lenguaje máquina son de 1 a 1, si bien existen algunas excepciones, que dan lugar a lo que se conoce como lenguajes macro-ensambladores [CUEVA 88]

- **Lenguajes Compilados**: Son aquellos, que son traducidos de un lenguaje de alto nivel (como FORTRAN o PASCAL) a lenguaje máquina o bien a lenguaje ensamblador, produciendo un programa objeto permanente.

- **Lenguajes Interpretados**: Estos lenguajes, tienen la particularidad, de que no producen código objeto, sino que cada instrucción es analizada y ejecutada a la vez, lo que ofrece mucha interacción con los usuarios, pero a la vez resultan ineficientes, cuando se desea ejecutar repetitivamente un programa.

- **Lenguajes Preprocesados**: Son lenguajes que son traducidos primeramente a un lenguaje intermedio de más bajo nivel, para posteriormente volverlos a traducir y producir el programa objeto. Este tipo de lenguajes fueron creados, con la idea de proporcionar un lenguaje más potente que el lenguaje intermedio, mediante la implementación de algunas macroinstrucciones. [SANCHIS-GALAN 86].

UNIDAD Nº 1
EL PARADIGMA ORIENTADO A OBJETOS
PARTE TEORICA

INTRODUCCION

La orientación a objetos es una forma natural de pensar en relación con el mundo y de escribir programas de computación. Mire a su alrededor. Por todas partes: *¡objetos!* Personas, animales, plantas, automóviles, aviones, edificios, cortadoras de pastos, computadoras y demás. Cada una de ellas tiene ciertas características y se comporta de una manera determinada. Si las conocemos, es porque tenemos el ***concepto de lo que son***. Conceptos persona, objetos persona. Los seres humanos **pensamos en términos de objetos**. Tenemos la capacidad maravillosa de la ***abstracción***, que nos permite ver una imagen en pantalla como personas, aviones, árboles y montañas, en vez de puntos individuales de color.

Todos estos objetos tienen algunas cosas en común. **Todos tienen atributos**, como tamaño, forma, color, peso y demás. Todos ellos exhiben **algún comportamiento.** Un automóvil acelera, frena, gira, etcétera. El objeto persona habla, ríe, estudia, baila, canta ...

Los seres humanos aprenden lo relacionado con los objetos estudiando sus atributos y observando su comportamiento. Objetos diferentes pueden tener muchos atributos iguales y mostrar comportamientos similares. Se pueden hacer comparaciones, por ejemplo, entre bebes y adultos, entre personas y chimpancés. Automóviles, camiones, pequeños autos rojos y patines tienen mucho en común.

La ***programación orientada a objetos*** *(POO)* **hace modelos de los objetos del mundo real** mediante sus contrapartes en software. Aprovecha las relaciones de clase, donde objetos de una cierta clase, como la clase de vehículos, tienen las mismas características. Aprovecha las relaciones de *herencia,* donde clases recién creadas de objetos se derivan heredando características de clases existentes, pero también poseyendo características propias de ellos mismos. Los bebes tienen muchas características de sus padres, pero ocasionalmente padres de baja estatura tienen hijos altos.

La programación orientada a objetos nos proporciona una forma más **natural e intuitiva** de observar el proceso de programación, es decir *haciendo modelos* de objetos del mundo real, de sus atributos y de sus comportamientos. POO también hace modelos de la comunicación entre los objetos. De la misma forma que las personas se envían *mensajes* uno al otro los objetos también se comunican mediante mensajes.

La **POO *encapsula* datos (atributos) y funciones (comportamiento)** en paquetes llamados ***objetos***; los datos y las funciones de un objeto están muy unidos. Los objetos tienen la propiedad de *ocultar* la *información.* Esto significa que aunque los objetos puedan saber cómo comunicarse unos con otros mediante *interfaces* bien definidas, a los objetos por lo regular no se les está permitido saber cómo funcionan otros objetos. Los detalles de puesta en práctica quedan ocultos dentro de los objetos mismos. (*Casi estamos diciendo que existe entre ellos el respeto a la intimidad ...*) A esto se le llama ***Encapsulamiento***.

En resumen, la programación Orientada a Objetos es un método de implementación en que los programas se organizan como colecciones de objetos que colaboran entre sí enviándose mensajes.

Solo disponemos de "***objetos que colaboran entre sí***".

Escenario: Mariano es alumno de la UTN y tiene que rendir el examen de PPR,

Javier, es el docente que lo evaluará.

COMPONENTES BASICOS

1- Objeto

Es una abstracción que utilizaremos como representación computacional de una entidad de la realidad. Tiene unas propiedades particulares (atributos) y una forma de operar con ellas (métodos). Un objeto maneja sus atributos mediante sus métodos.

Ejemplo: del escenario antes planteado las entidades que debemos modelizar para representar esa realidad son: Mariano, Javier y el examen.

Si consideremos el objeto examen, sus atributos serán fecha, materia, enunciado, resolución, puntuación, nota, alumno que rindío y docente que evalúo; y los métodos o acciones que podemos hacer con ellos pueden ser leer el enunciado, hacer resolución, consultar puntuación, asignar nota, consultar nota, etc.

Todo objeto tiene **estado**, exhibe un **comportamiento** bien definido y posee una **identidad** única.

Estado: abarca todas las propiedades (normalmente estáticas) del mismo más los valores actuales (normalmente dinámicas) de cada una de estas propiedades.

Comportamiento: es como actúa y reacciona un objeto en términos de sus cambios de estado y paso de mensajes.

Identidad: es aquella propiedad de un objeto que lo distingue de todos los demás. Permite identificar un objeto dentro de un conjunto de objetos.

Creación de un Objeto

Partiendo del hecho de que en el Paradigma Orientado a Objetos solo disponemos de objetos, llegamos a la conclusión, de que para crear un objeto es necesario contar con otro objeto que tenga la responsabilidad, entre otras, de crear objetos.

El objeto creador de objetos se conoce con el nombre de **clase** y, los objetos creados se denominan **instancia** o ejemplares de clase.

Una **clase** es simplemente un modelo que se utiliza para describir a objetos similares. Para ello, define los atributos y métodos de los objetos instancia de ella.

La clase define el método que ejecutará el objeto al recibir un determinado mensaje consecuentemente, todos los objetos de una clase dada usan el mismo método en respuesta a mensajes similares.

En nuestro escenario, el objeto creador del objeto Mariano es la clase Alumno, el de Javier, es la clase Docente y el del examen de PPR es Examen.

Por ejemplo, las clases podrían tener lo siguiente:

Clase Examen

Atributos: fecha, materia, enunciado, resolución, puntuación, nota, alumno y docente**.**

Métodos: leer el enunciado, hacer resolución, consultar puntuación, asignar nota, consultar nota, etc.

Clase Alumno

Atributos: legajo, nombre, dni, domicilio, curso.

Métodos: mostrar legajo, mostrar nombre, mostrar dni, mostrar domicilio, mostrar curso, cambiar domicilio, cambiar curso.

Clase Docente

Atributos: legajo, nombre, dni, domicilio, materia que dicta.

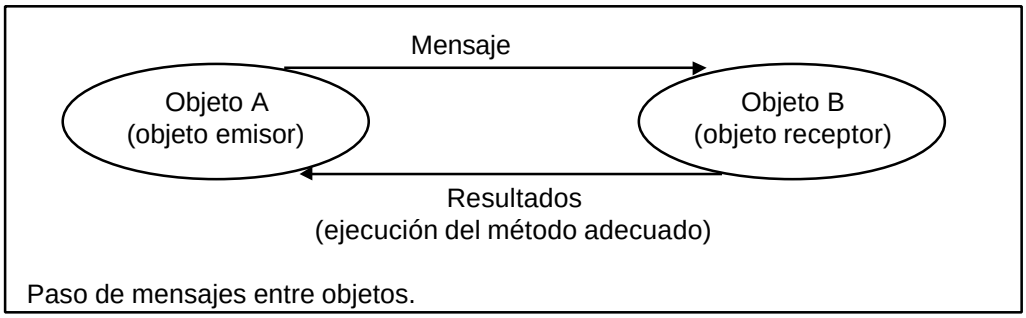
Métodos: mostrar legajo, mostrar nombre, mostrar dni, mostrar domicilio, mostrar materia, cambiar domicilio, cambiar materia.

Destrucción de un Objeto

Con este marco conceptual, se deduce que un objeto se destruye cuando recibe un mensaje de otro objeto que solicite ese comportamiento, siempre y cuando, el estado de este objeto así lo permita, en otras palabras, siempre que no este colaborando con otros objetos. Esto implica, que todo objeto debe poseer un método que efectivice su destrucción.

2- Mensajes

Un mensaje es una petición de un objeto a otro para que éste se comporte de una determinada manera, ejecutando uno de sus métodos. La técnica de enviar mensajes se conoce como *Paso de Mensajes*.



Paso de mensajes entre objetos.

Los mensajes relacionan un objeto con otros y con el mundo exterior. Propician la colaboración entre objetos.

El objeto receptor del mensaje no sabe quien le envío el mensaje (no conoce al objeto emisor) solo se comporta según el mensaje recibido.

El conjunto de mensajes a los cuales puede responder un objeto se le conoce como ***protocolo del objeto***.

El protocolo del objeto es público, mientras que la definición de sus datos y métodos es privada, es interno al objeto. Esta parte interna es la implementación y se encuentra encapsulada.

CARACTERISTICAS DE LA PROGRAMACION ORIENTADA A OBJETOS

Para todo lo orientado a objetos el marco referencial conceptual es el Modelo de Objetos. Existen 4 elementos fundamentales en este modelo:

1. Abstracción
2. Encapsulamiento
3. Herencia
4. Polimorfismo

Al decir fundamentales quiere decir que un modelo que carezca de cualquiera de estos elementos no es orientado a objetos.

Existen otras características, que son secundarias, y que varían dependiendo de los autores que las enuncian, por ejemplo Grady Booch en su libro “Análisis y diseño orientado

a objetos con aplicaciones" enuncia tres características secundarias: tipificación, concurrencia y persistencia.

1. Abstracción

Es una descripción simplificada o especificación de un sistema que enfatiza algunos de los detalles o propiedades del mismo mientras suprime otros. Una buena abstracción es aquella que enfatiza detalles significativos al lector o usuario y suprime detalles que son, al menos por el momento, irrelevantes o causa de distracción. Consecuentemente, dicha abstracción dependerá de la perspectiva del observador.

Por ejemplo, en nuestro escenario la entidad alumno es una persona y como tal tiene caracteristicas personales tales como peso, estatura, contextura fisica, etc. Pero que en nuestro modelo son irrelevantes ya que para estudiar o aprobar las materias no es necesaria su apariencia fisica. Distinto es el caso de un modelo publicitario, por ejemplo, donde esas caraterísticas, de apariencia física, son relevantes para desempeñar cualquier trabajo que se le encomiende.

1. Encapsulamiento

El encapsulamiento hace referencia a la ocultación de información concerniente a la estructura de un objeto y a la implementación de sus métodos.

Puede decirse que todos los objetos de un sistema encapsulan algún estado y que todo el estado de un sistema esta encapsulado en objetos. Esto nos lleva a la conclusión de que el estado del sistema esta determinado por el estado de todos los objetos que lo modelan.

Un objeto es percibido por los demás objetos como una caja negra. Ahora bien, el protocolo de mensajes debe ser conocido por todos los objetos para que puedan colaborar entre si, manteniéndose ocultas las acciones que ejecuta el objeto al recibir un mensaje específico.

La abstracción y el encapsulamiento son complementarios: la abstracción se centra en el comportamiento observable de un objeto; mientras que el encapsulamiento se centra en la implementación que da lugar a este comportamiento.

En nuestro escenario, el objeto Javier envía el mensaje "entregar examen a Mariano" al objeto Mariano; el objeto Javier no conoce las acciones que desarrolla el objeto Mariano para realizar el examen pero, tampoco le interesan ya que lo único que le importa es que el objeto Mariano realice el examen.

2. Herencia

Es la propiedad que permite a los objetos construirse a partir de otros objetos. Es la ordenación de diferentes niveles de abstracción.

En el mundo real la herencia impone una jerarquía entre clases. El principio de este tipo de división es que cada subclase comparte características comunes con la clase de la que se deriva. Además de las características compartidas con otros objetos de la clase, cada subclase tiene sus propias características particulares.

La herencia establece una relación jerárquica entre clases en la cual una clase hija (subclase) hereda de su clase padre (base). Si una clase solo puede recibir atributos y métodos de una sola clase se denomina herencia simple, si recibe atributos y métodos de más de una clase base se denomina herencia múltiple.

En nuestro escenario, Mariano (el alumno) y Javier (el docente) tienen caracteristicas comunes: nombre, dni y domicilio, porque son características que posee cualquier persona.

Por lo tanto podríamos agrupar esas características, y formar una nueva clase, llamada Persona que tenga las características comunes a cualquier persona.

Clase Persona

Atributos: nombre, dni, domicilio.

Métodos: mostrar nombre, mostrar dni, mostrar domicilio, cambiar domicilio.

Como ya tenemos estos datos básicos, ahora podemos rediseñar las clases alumno y docente:

El alumno como **es una** persona que **tiene** todos los datos de persona y además los datos propios de alumno, por lo tanto quedaría conformada de la siguiente forma:

Clase Alumno (es una persona)

Atributos: Atributos de persona, legajo, curso.

Métodos: Métodos de persona, mostrar legajo, mostrar curso, cambiar curso.

El docente tambíen **es una** persona, por lo tanto **tiene** todos los datos de persona y además sus propios datos como docente, por lo tanto quedaría conformada de la siguiente forma:

Clase Docente (es una persona)

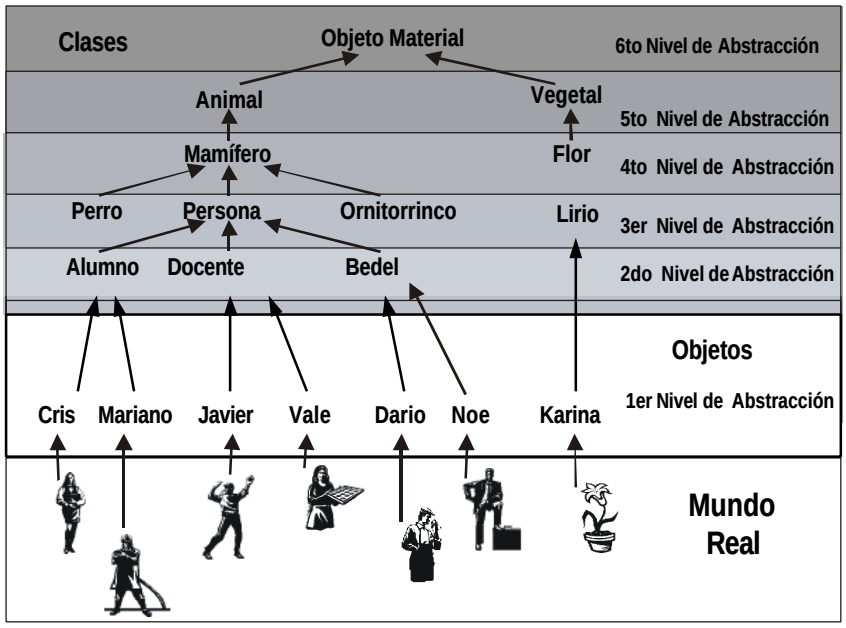
Atributos: Atributos de persona, legajo, materia que dicta.

Métodos: Métodos de persona, mostrar legajo, mostrar materia, cambiar materia.

Este nuevo diseño nos muestra que podemos implementar la clase persona una vez y usarla en alumno y docente, esto logra la reutilización ya que se ahorra tiempo de implementación, también se ahorra tiempo en el control de errores de codificación y depuración, ya que si se tienen rutinas ya probadas y funcionando no hay que volver a controlarlas.

Además nos permite hacer los programas extensibles, es decir, si en un futuro se agregara la entidad Bedel, por ejemplo, solo la extendemos (heredamos) de persona ya que es una persona.

Analicemos la siguiente representación gráfica de nuestro escenario (estructura jerárquica de clases).



En la figura anterior, la forma de pensar en cómo se ha organizado el conocimiento acerca de Mariano es en términos de una jerarquía de clases. Mariano **es un** Alumno (clase), pero Alumno **es una** forma especializada de Persona (clase); Persona **es** también **un** Mamífero (clase); así se sabe, por ejemplo, que Mariano **es un** mamífero.

De esta manera, gran parte del conocimiento que se tiene y que es aplicable también a la categoría más específica se llama herencia. Se dice, por ejemplo, que Alumno heredará atributos y métodos de la clase Persona.

La regla más importante para saber cuando una clase debe convertirse en subclase de otra o cuándo es más apropiado otro mecanismo es que, para que una clase se relacione con otra por medio de la herencia, debe haber una relación de funcionalidad entre ambas. Entonces describimos la regla es-un para captar dicha relación.

Por otra parte, si consideramos que en el Paradigma OO solo disponemos de objetos es lógico afirmar que los componentes de objetos son objetos. Esto establece la relación tiene-un.

4. Polimorfismo

En un sentido literal, polimorfismo significa “cualidad de tener más de una forma”. En el contexto de la POO, el polimorfismo se refiere al hecho de que un mismo mensaje puede generar diferentes comportamientos en diferentes objetos. En otras palabras, reaccionan al mismo mensaje de modo diferente.

Un ejemplo podría ser el siguiente: un usuario de un sistema tiene que imprimir un listado de sus clientes, puede elegir imprimirlo en el monitor, en la impresora, en un archivo en disco o en una terminal remota.

De esta situación identificamos los objetos monitor, impresora, archivo y terminal remota. Todos estos objetos pueden recibir el mensaje “imprimir listado”, y de acuerdo al objeto de que se trate se activará el comportamiento correspondiente.

VENTAJAS

Algunas de las principales ventajas que incorpora la Programación Orientada a Objetos en el desarrollo de software son: modelos más naturales, modularidad, extensibilidad, eliminación de redundancias, facilidad de reutilización.

MODELOS

En POO es posible construir un modelo del sistema que sea casi independiente de los requisitos del proyecto. La razón es que en POO son los datos los que establecen la jerarquía que en la programación clásica viene definida por los programas. Esta inversión hace que los modelos se establezcan de forma similar a la manera de razonar del hombre y, por ende, de forma mucho más natural.

Cuando un ser humano analiza un programa o una situación, suele trabajar con conceptos (modelos mentales de objetos) y relaciones entre ellos, y sólo secundariamente acude a la funcionalidad de los mismos. Esto significa que, en la programación clásica por ejemplo, es preciso traducir el modelo mental de objetos jerarquizados a uno más práctico de programas y subrutinas. En el caso de la POO, dicha traducción es innecesaria, pues el modelo mental puede constituir, en sí mismo, la base de organización de la aplicación POO. La eliminación de este paso intermedio de traducción puede tener consecuencias favorables para la reducción de los errores iniciales de implementación.

MODULARIDAD

La modularidad es uno de los conceptos clave en la evaluación del software y uno de los principales objetivos de los diseñadores de aplicaciones. Un programa es modular si se compone de módulos independientes y robustos, lo que lleva a una mayor facilidad de

reutilización de componentes, así como a la verificación o depuración de los mismos.

La modularidad en la POO es mucho más intuitiva que en la programación tradicional. El concepto de módulo está directamente relacionado con el objeto. Los objetos son módulos naturales, que corresponden a una imagen lógica de la realidad.

EXTENSIBILIDAD

Durante el desarrollo de sistemas, la aparición de nuevos requisitos puede ser problemática, por eso es deseable que las herramientas de desarrollo nos permitan añadir esos nuevos requisitos sin modificar la estructura básica de nuestro diseño. Un sistema POO bien diseñado puede lograr esto, que si bien se trata de una técnica que aporta muchas posibilidades y que se adapta especialmente bien en ciertos desarrollos, también ocurre como con cualquier otra técnica, es preciso aplicarla con las debidas precauciones.

Para que un sistema POO esté bien diseñado, es importante en primer lugar que el modelo sea correcto, es decir, que se haya definido de forma adecuada la jerarquía de las clases, así como los atributos y métodos asociados, que describen nuestro problema. En POO, el análisis del problema es la fase más importante y crítica de todo el ciclo de vida del software. Un modelo bien construido nos permitirá, sin ninguna complicación añadir:

- Nuevos casos particulares de las clases preexistentes.
- Nuevas clases y subclases.
- Nuevas propiedades.
- Conductas o métodos nuevos.

ELIMINACION DE REDUNDANCIAS

En un desarrollo de sistemas se desea evitar la definición múltiple de datos o funciones comunes, se intenta conseguir que cada elemento tenga que ser definido una vez.

Entre las características de la POO hemos mencionado la herencia. Esta propiedad es la clave que permite la eliminación de redundancias, pues evita la definición múltiple de propiedades comunes a muchos objetos. En POO, las propiedades o atributos se definen en el antepasado más lejano que las comparta. Lo mismo sucede con los métodos, que reflejan acciones y que son heredados por todos los descendientes bajo la presunción de que se compartan de la misma forma, aunque dicho comportamiento puede modificarse, en ciertos casos particulares, en virtud del polimorfismo.

REUTILIZACION

En los proyectos de desarrollo, a menudo se multiplica el esfuerzo requerido para su programación porque se repite innecesariamente la codificación de módulos que realizan funciones sencillas. Todo programador ha codificado repetidas veces rutinas que, en el fondo, tienen la misma funcionalidad. También puede suceder que varias personas inviertan su esfuerzo simultáneamente en programas equivalentes. Todo esto podría evitarse si cada programa, una vez codificado, pudiera utilizarse tantas veces como sea necesario en todas aquellas aplicaciones que precisen de su funcionalidad.

La POO proporciona un marco perfecto para la reutilización de las clases. El encapsulamiento y la modularidad nos permiten utilizar una y otra vez las mismas clases en aplicaciones distintas. En efecto, el aislamiento entre las distintas clases significa que es posible añadir una clase o un módulo nuevo (extensibilidad) sin afectar al resto de la aplicación. Por tanto, no hay nada que se oponga a que dicha clase o módulo sea exactamente la misma que se ha utilizado en una aplicación distinta.

La reutilización de los objetos es uno de los motivos principales que justifican la utilización de la metodología orientada a objetos en la mayor parte de los casos. Por eso, cualquier sistema comercial de POO suele venir provisto de un conjunto de clases predefinidas, lo que permite ahorrar tiempo y esfuerzos en el desarrollo de las aplicaciones.

DESVENTAJAS

Si bien la POO es una forma de trabajar que aporta importantes innovaciones técnicas respecto a métodos anteriores de desarrollo, estas técnicas no nos proporcionan una panacea que reduzca a cero el esfuerzo de programación. Es preciso seguir trabajando mucho para realizar un proyecto, aunque varía la distribución del esfuerzo entre las distintas fases del desarrollo.

Tampoco hay que pensar que podemos olvidarnos de los principios clásicos y establecidos de la ingeniería del software. La programación orientada a objetos aporta innovaciones importantes, pero también exige hacer un uso abundante y sesudo de dichos principios. Las peculiaridades de la metodología OOP en sus distintas fases (análisis, diseño, codificación, pruebas, mantenimiento) respetan dichos principios. Es necesario tener esto muy presente antes de afrontar cualquier proyecto serio.

Por otro lado, la programación orientada a objetos es una filosofía de programación que, lejos de requerir una capacidad menor por parte de los programadores, tiende a agrandar la diferencia entre los buenos y los malos. Es una técnica atractiva, pero también peligrosa entre las manos de un programador incompetente. Es necesario comprender y, sobre todo, asimilar los fundamentos de la POO. La experiencia de los autores indica que los vicios propios de la programación clásica persisten y son el mayor obstáculo inicial que hay que salvar antes de encontrarse como el pez en el agua en el desarrollo de software orientado a objetos.

Tampoco sería correcto sacar la conclusión de que la programación orientada a objetos es una técnica incompatible con todas las demás. Más bien al contrario, hay lenguajes híbridos que pueden incorporar ambas técnicas de programación, estructurada y orientada a objetos, como vimos anteriormente.

LENGUAJES DE PROGRAMACION ORIENTADOS A OBJETOS

El lenguaje de programación orientado a objetos nace oficialmente en Noruega (1967) con el lenguaje SIMULA-67, destinado a realizar simulaciones de objetos discretos. Este es un lenguaje de propósito general que incluye conceptos como encapsulamiento, herencia, etc. aunque, sigue siendo un lenguaje convencional: tipear el archivo fuente, compilarlo, editarlo, ejecutarlo y, se repite el ciclo cada vez que sea necesario.

De todas maneras, se lo considera como el precursor de la tecnología de objetos tal cual se la conoce actualmente.

Sus inicios ...

Año 1969, ALAN KAY (personaje fundamental de la informática) estudió como mejorar el proceso cognitivo utilizando herramientas informáticas. Para ello se contactó con PAPPERT (desarrollador del lenguaje LOGO) y llegaron a la conclusión de que los niños alcanzaban conceptualizaciones abstractas complejas jugando con Logo.

En esta década, de los mainframes, KAY se pregunto qué computadora necesitaría para realizar este mejoramiento del proceso cognitivo y definió conceptualmente “DynaBook” (computadora portátil, semejante a una agenda, que permitiera al usuario disponer de todo lo que necesite y conectarse con otras personas que tuvieran DynaBook). “Algo” como las Palmtop de hoy y aún mucho mejor.

En el año 1970, la empresa XEROX funda el laboratorio PARC con el objetivo de desarrollar tecnología de punta sin fines de lucro inmediatos (no comercializa directamente las tecnologías que produce) y contratan a Alan KAY para que construya su DynaBook.

Tengamos presente que en esa época, la tecnología solo permitía pensar (como mucho) en las computadoras de escritorio. KAY pensando en que sea fácil de usar para los niños presta especial atención en la interfaz diseñando de esta manera el concepto de

"interfaz gráfica". Se decide utilizar el sistema operativo UNIX como base ya que en esos momentos era ampliamente difundido y utilizado en las universidades de EEUU, naciendo así, la primera máquina WorkStation Unix.

El siguiente paso, era generar un proyecto de software que debería ser fácil de usar y entender. Lideró este proyecto Adele GOLDBERG, quien sobre Unix y tomando ideas de SIMULA construyeron: SMALLTALK, la primera implementación de las ideas de Alan Kay.

En 1972, se dispuso de la versión académica: sistema operativo – interfaz gráfica – entorno de desarrollo. Luego el grupo entregó SmallTalk a empresas, universidades y laboratorios; saliendo una nueva versión en 1976.

En 1980, X-PARC da por terminado el proyecto de Smalltalk y Adele Goldberg funda una empresa para comercializarlo como entorno de desarrollo para empresas grandes, constituyéndose en el único ambiente de objetos puro que ha tenido difusión comercial.

Sus características son:

- Trabaja en un ambiente de objetos.
- Ofrece una interfaz gráfica.
- Permite interactuar fácilmente con el programa que se esta ejecutando.
- Rompe el ciclo convencional en la construcción de programas(tipeado, compilado,...).

A partir de aquí comienzan a haber distintas vertientes de objetos:

C++ Bjórn Stroutrop: lenguaje orientado a objetos montado sobre el lenguaje C para aprovechar que en ese momento la mayor parte de los programadores eran expertos en C. El lenguaje tuvo un gran éxito pero no dejo de ser una "evolución" del lenguaje C y no una "revolución" que implicaría la utilización del paradigmas de objetos puro.

En esta misma línea aparece Java. Ambos, C++ y Java, siguen siendo ambientes Híbridos.

Bertrand Meye: enfocó el paradigma orientado a objetos desde la teoría d ela computación. Dando origen a EIFFEL, más orientado a objetos que C++ pero menos que Smalltalk, pero incorporando la ingeniería de software: software de alta calidad y seguro.

Brad Cook: desarrolla Objetive C (tomado más cosas de Smalltalk que de C). Su idea era desarrollar software como los electrónicos desarrollan lo suyo.

Luego surgieron los metodólogos como Jourdon, Jacobson, Booch, Cod y Rebecca Wirfs-Brock.

Importante

El Paradigma de Programación Orientado a Objetos plantea un modelado de la realidad que reviste una complejidad no soportada completamente por algunos lenguajes de programación.

De ahí la clasificación en:

Lenguajes puros: los que solo permiten realizar programación orientada a objetos. Estos incluyen Smalltalk, Eiffel, HyperTalk y Actor.

Lenguajes Híbridos: los que permiten mezclar programación orientada a objetos con la programación estructurada básica. Estos incluyen C++, Objetive C, Objetive Pascal, Java.

Debemos tener en cuenta que si no consideramos las características de la POO puede programarse en un lenguaje como Objetive Pascal, C++, Java, pero el diseño tendrá el aspecto de una aplicación Fortran, Pascal o C.

PROGRAMACION ORIENTADA A OBJETOS EN JAVA

INTRODUCCION

En esta sección explicaremos la POO desde el lenguaje Java.

Java es un lenguaje totalmente basado en clases y objetos. Para comprender realmente java, debemos entender estos conceptos, ya que los encontraremos en cada pequeño programa o gran aplicación que construyamos.

En los lenguajes normales existen los datos y las funciones o rutinas que operan sobre esos datos. Pero los datos son entidades de alguna forma totalmente separadas de las funciones que operan sobre ellos. En la tecnología de objetos, en cambio, una clase es un plano para un objeto, que es, en definitiva, una entidad que tiene sus propios datos y funciones asociadas. Mientras que en la programación tradicional podemos encontrarnos con los datos de un empleado, por un lado, y las funciones que operan sobre esos datos, por otro, en un lenguaje orientado a objetos definimos la clase Empleado, que contendrá tanto los datos que describen a un empleado, como las funciones y rutinas que pueden aplicarse a esos datos.

Esto permite manejar entidades, descriptas en clases, que contienen tanto la información de su estado (datos) como su conducta (métodos). En la programación orientada a objetos, uno termina definiendo cosas, que son descriptas en clases. Esta es una de las razones de la adopción de esta disciplina: Nos facilita las descripción de las entidades del mundo real. Además la construcción de una clase, la definición de sus datos y especificación de sus métodos, nos lleva a pensar el sistema modularmente. Cada clase puede diseñarse e implementarse independientemente de las demás. Esto también facilita la utilización de una clase en otro proyecto, y la modificación de alguna en el tiempo, sin necesidad de modificar el resto de las clases. Se dice que las clases posibilitan la utilización repetida y el mantenimiento del código escrito.

CLASES

INTRODUCCION

Como ya dijimos, una clase es un modelo que se utiliza para describir a objetos similares.

Las clases son lo más simple de Java. Todo en Java forma parte de una clase, es una clase o describe como funciona una clase. El conocimiento de las clases es fundamental para poder entender los programas Java.

Todas las acciones de los programas Java se colocan dentro del bloque de una clase o un objeto. Todos los métodos se definen dentro del bloque de la clase, Java no soporta funciones o variables globales. Esto puede despistar a los programadores de C++, que pueden definir métodos fuera del bloque de la clase, pero esta posibilidad es más un intento de no separarse mucho y ser compatible con C, que un buen diseño orientado a objetos. Así pues, el esqueleto de cualquier aplicación Java se basa en la definición de una clase.

Todos los datos básicos, como los enteros, se deben declarar en las clases antes de hacer uso de ellos. En C la unidad fundamental son los archivos con código fuente, en Java son las clases. De hecho son pocas las sentencias que se pueden colocar fuera del bloque de una clase. La palabra clave import (equivalente al #include) puede colocarse al principio de un archivo, fuera del bloque de la clase. Sin embargo, el compilador reemplazará esa sentencia con el contenido del archivo que se indique, que consistirá, como es de suponer, en más clases.

Una clase es un marco que permite crear objetos de su misma estructura. Estos objetos constan de:

- Variables de clase y de instancia, que son los descriptores de atributos y entidades de los objetos.
- Métodos, que definen las operaciones que pueden realizar esos objetos.

Un objeto se instancia a partir de la descripción de una clase: un objeto es dinámico, y puede tener una infinidad de objetos descritos por una misma clase. Un objeto es un conjunto de datos presente en memoria.

ESTRUCTURA GENERAL

Una clase contiene elementos, llamados miembros, que pueden ser datos, llamados *atributos,* y funciones que manipulan esos datos llamados *métodos*.

Una clase se define con la palabra reservada class.

La sintaxis de una clase es:

```
[public] [final | abstract] class nombre_de_la_Clase [extends ClaseMadre]
        [implements Interfase1 [, Interfase2 ]...]
{
        [Lista_de_atributos]
        [lista_de_métodos]
}
```

Todo lo que está entre [y] es opcional. Como se ve, lo único obligatorio es class y el nombre de la clase.

Public, final, abstract

Definir una clase como pública (**public**) significa que puede ser usada por cualquier clase en cualquier paquete. Si no lo es, solamente puede ser utilizada por clases del mismo paquete (un paquete, básicamente, es un grupo de clases e interfaces relacionadas, como los paquetes de biblioteca incluidos con Java).

Una clase final (**final**) es aquella que no puede tener clases que la hereden. Esto se utiliza básicamente por razones de seguridad (para que una clase no pueda ser reemplazada por otra que la herede), o por diseño de la aplicación.

Una clase abstracta (**abstract**) es una clase que puede tener derivadas, pero no puede ser instanciada. Es literalmente abstracta. ¿Para qué sirve? Para modelar conceptos. Por ejemplo, la clase Number es una clase abstracta que representa cualquier tipo de números (y sus métodos no están implementados: son abstractos); las clases descendientes de ésta, como Integer o Float, sí implementan los métodos de la madre Number, y se pueden instanciar.

Por todo lo dicho, una clase no puede ser final y abstract a la vez (ya que la clase abstract requiere descendientes).

Extends

La instrucción extends indica de qué clase desciende la nuestra. Si se omite, Java asume que desciende de la superclase **object**.

Cuando una clase desciende de otra, significa que hereda sus atributos y sus métodos. Esto quiere decir que, a menos que los redefinamos, sus métodos serán los mismos que los de la clase madre y podrán utilizarse en forma transparente, siempre y cuando no sean *privados* en la clase madre, o protegidos o propios del paquete para

subclases de otros paquetes.

DECLARACION Y DEFINICION

La *declaración* lista los miembros de la clase. La definición, también llamada *implementación*, define las funciones de la clase.

La declaración y la definición de la clase van juntas. Por ejemplo:

```
//Implementación de un contador sencillo
//GRABAR EN UN ARCHIVO "Contador.java"
// COMPILAR CON: "javac Contador.Java
public class Contador {   //     Se declara y define la clase Contador
      int cnt;
      public void Inicializa() {
            cnt=0;       //inicializa en 0 la variable cnt
      }
      //Otros métodos
      public int incCuenta() {
            cnt++;
            return cnt;
      }
      public int getCuenta() {
            return cnt;
      }
}
```

EL CUERPO DE LA CLASE

El cuerpo de la clase, encerrado entre { y }, es la lista de atributos (variables) y métodos (funciones) que la constituyen. No es obligatorio, pero en general se listan primero los atributos y luego los métodos.

INSTANCIAS DE UNA CLASE (OBJETOS)

Los objetos de una clase son instancias de la misma. Se crean en tiempo de ejecución con la estructura definida en la clase.

Para crear un objeto de una clase se usa la palabra reservada new.

Por ejemplo si tenemos la siguiente clase:

```
public class Cliente {
      private int codigo;
      private float importe;
      public int   getCodigo()  { return codigo; }
      public float getImporte() { return importe; }
      public void setImporte(float x) { importe = x; }
};
```

el objeto o instancia de la clase cliente es:

```
Cliente comprador = new Cliente();      //objeto o instancia de la clase
                                        //Cliente
```

El operador new reserva espacio en memoria para los miembros dato y devuelve una referencia que se guarda en la variable comprador de tipo Cliente que denominamos ahora objeto. Dicha sentencia, crea un objeto denominado comprador de la clase Cliente.

Implementando el ejemplo en forma completa:

```
// Archivo:  ManejaCliente.java
//Compilar con: javac ManejaCliente.java
//Ejecutar con: java ManejaCliente
import java.io.*;

Public class ManejaCliente {
      //el punto de entrada del programa
```

```
        public static void main(String args[]) {
                Cliente comprador = new Cliente();      //crea un cliente

                comprador.setImporte(100);       //asigna el importe 100

                float adeuda = comprador.getImporte();
                System.out.println("El importe adeudado es "+adeuda);
        }
}
```

Acceso a los miembros

Desde un objeto se puede acceder a los miembros mediante la siguiente sintaxis

objeto.miembro;

Por ejemplo, podemos acceder al método setImporte, para cambiar el importe de la deuda de un objeto cliente.

```
comprador.setImporte(100);
```

Si el cliente comprador, por ejemplo, tenía inicialmente un importe de 0, mediante esta sentencia se lo cambiamos a 100.

Desde un objeto llamamos a las funciones miembro para realizar una determinada tarea. Por ejemplo, desde el cliente comprador llamamos a la función getImporte() para mostrar el importe de dicho cliente.

```
Comprador.getImporte();
```

La función miembro getImporte() devuelve un número, que guardaremos en una variable adeuda, para luego usar este dato.

```
float adeuda=comprador.getImporte();
System.out.println("El importe adeudado es "+adeuda);
```

Como veremos mas adelante, no siempre es posible acceder a los miembros, el acceso dependerá de los controles de acceso a los mismos.

Ciclo de Vida de los Objetos

Cuando se ejecuta un programa orientado a objetos ocurren tres sucesos.

1. Los objetos se crean a medida que se necesitan.
2. Los mensajes se mueven de un objeto a otro (o del usuario a un objeto) a medida que el programa procesa información o responde a la entrada del usuario.
3. Cuando los objetos ya no se necesitan, se borran y se libera la memoria.

La vida de un objeto

En el lenguaje C++, los objetos que se crean con new se han de eliminar con delete. new reserva espacio en memoria para el objeto y delete libera dicha memoria. En el lenguaje Java no es necesario liberar la memoria reservada, el recolector de basura (garbage collector) se encarga de hacerlo por nosotros, liberando al programador de una de las tareas que más quebraderos de cabeza le producen, olvidarse de liberar la memoria reservada.

Arrays de objetos

Se puede crear un array de objetos de la misma forma que se crea un array de

elementos de otro tipo. Por ejemplo:

En general, un arreglo se declara de la siguiente forma:

```
Cliente compradores [] = new Cliente[100];
```

Ahora bien, como se trata de un arreglo de objetos, los que esta sentencia está creando son 100 referencias a un Cliente y no 100 Clientes, para poder tener 100 clientes es necesario crearlos a cada uno usando esas referencias:

```
for( int i=0; i < 100; i++ )
    compradores[i] = new Cliente();
```

Con este código, recién se están creando los 100 clientes en el arreglo.

Implementando el ejemplo en forma completa tenemos:

```
//Implementación de la clase cliente
//GRABAR EN UN ARCHIVO "Cliente.java"
// COMPILAR CON: "javac Cliente.Java

public class Cliente {
       private int codigo;
       private float importe;
       public int   getCodigo()  { return codigo; }
       public float getImporte() { return importe; }
       public void setImporte(float x) { importe = x; }
};
```

Para probar esta clase (mínima) con el siguiente ejemplo de aplicación:

```
// Archivo:  ManejaClientes.java
//Compilar con: javac ManejaClientes.java
//Ejecutar con: java ManejaClientes
import java.io.*;

Public class ManejaClientes {
       //el punto de entrada del programa
       public static void main(String args[]) {
              Cliente comprador = new Cliente();      //crea un cliente

              comprador.setImporte(100);       //asigna el importe 100

              float adeuda = comprador.getImporte();
              System.out.println("El importe adeudado es "+adeuda);

              //usando un arreglo de objetos Cliente
              Cliente compradores [] = new Cliente[100];

              for( int i=0; i < 100; i++ )
                   compradores[i] = new Cliente();   //crea los clientes

              for (int i=0; i<100;i++)
                     compradores[i].setImporte(0);   //inicializa los
                                                      //importes en 0
       }
}
```

DECLARACION DE MIEMBROS UNA CLASE

Los datos de una clase se denominan **atributos** y las funciones de una clase se denominan **métodos**.

Los miembros tienen ciertas restricciones en el modo en que se puede manipular los mismos dentro y fuera de la clase, a esto se le llama control de acceso a una clase o visibilidad.

Con estas restricciones se logra la encapsulación que, como vimos en la introducción, consiste en separar los aspectos externos del objeto, a los cuales pueden acceder otros objetos, de los detalles de implementación del mismo, que quedan ocultos para los demás.

La encapsulación se basa en la noción de servicios prestados; Una clase proporciona un cierto número de servicios y los usuarios de esta clase no tienen que conocer la forma como se prestan estos servicios. Contrariamente a las cuentas de una asociación caritativa, una clase debe mantenerse opaca.

Hay que distinguir pues en la descripción de la clase dos partes:

- la parte pública, accesible por las otras clases;
- la parte privada, accesible únicamente por los métodos de la clase.

Se recomienda encarecidamente poner los atributos de una clase en la parte privada, para respetar el concepto de encapsulamiento.

Tomando como ejemplo una clase Ventana, los métodos que modifican el tamaño de la ventana no deben manipular directamente los atributos del objeto. Admitamos, para el ejemplo, que estos atributos sean públicos. Por ejemplo tendríamos el siguiente código:

```
class Ventana {
      // la palabra public permite que los miembros siguientes sean de
      acceso publico
      public int x1, yl;        // coordenadas arriba izquierda
      public int h, anchura;    // altura. Anchura
};
```

Entonces se pueden modificar directamente estos atributos:

```
class Pantalla {
      public void DesplazaloEnDiagonal (Ventana unav) {
            unav.xl += l0;
            unav.yl += 10;
      }
};
```

Resulta práctico... pero muy imprudente. En efecto, supongamos que decide más adelante añadir los atributos siguientes a la clase Ventana:

```
int x2, y2; // coordenadas de la esquina inferior derecha
```

por ejemplo, para no tener que recalcularlas cada vez.

El método DesplazaloEnDiagonal debe modificarse para tener en cuenta estos nuevos atributos:

```
class Pantalla {
      public void DesplazaloEnDiagonal (Ventana unav) {
            unav.x1 += 10;
            unav.yl += 10;
            unav.x2 += 10;
            unav.y2 += 10;
      }
}
```

Es enojoso. Una modificación de la clase Ventana que no ofrece servicios suplementarios entraña entonces una modificación de las otras clases que utilizan Ventana. El mantenimiento de este programa se complica.

En realidad, se han cometido dos errores sucesivos en el ejemplo anterior:

- un error de encapsulamiento: los atributos x1 y x2 no debieran ser manipulables directamente;
- un error de diseño de método: el método DesplazaloEnDiagonal no debiera situarse en la clase Pantalla, sino en la clase Ventana:

```
class Ventana {
```

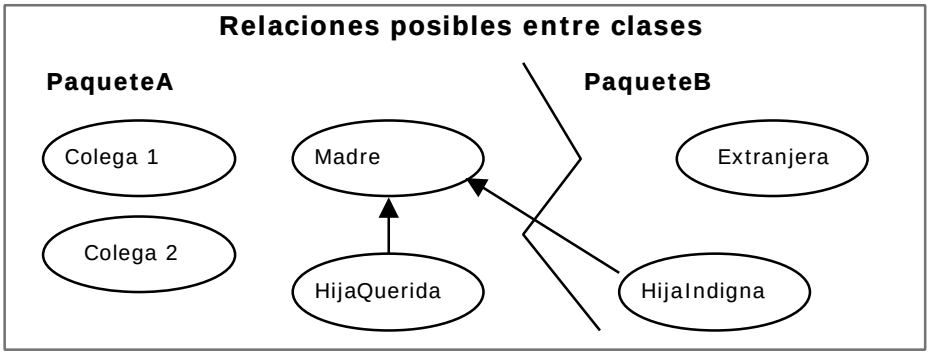
```
        // la palabra private permite que los miembros siguientes sean de
        acceso privado
        private  int x1;          // coordenadas arriba
        private int yl;           // coordenadas izquierda

        private int h;            // altura.
        private int anchura;      // Anchura
        private int x2;           // coordenadas de la esquina inferior
        private int y2;           // derecha
        public void DesplazaloEnDiagonal () {
              xl+=10;      yl+=l0;
              x2+=10;      y2+=10;
        }
};
```

Se constata además que la escritura resulta más simple. Especialmente, al escribirlo así, se acerca el método a los datos que manipula. Finalmente, añadir los atributos x2 e y2 modifica entonces un solo método de la clase Ventana, en lugar de modificar numerosos métodos de los usuarios de Ventana.

En resumen, en Java la unidad primigenia de encapsulado es la clase y como estén declarados sus miembros determina el grado de encapsulamiento. A continuación analizaremos en detalle los modificadores de acceso de los miembros de una clase:

Modificadores de acceso a miembros de clases

Java proporciona varios niveles de encapsulamiento que vamos a examinar sucesivamente en el dibujo siguiente.



Hemos representado en este dibujo una clase Madre alrededor de la cual gravitan otras clases:

- sus hijas HijaQuerida e HijaIndigna, la segunda de las cuales se encuentra en otro paquete; estas dos clases heredan de Madre. la herencia se explica en detalle algo más adelante, pero retenga que las clases HijaQuerida e HijaIndigna se parecen mucho a la clase Madre. Los paquetes se detallan igualmente algo más adelante; retenga que un paquete o *package* es un conjunto de clases relacionadas con un mismo tema destinada para su uso por terceros, de manera análoga a como otros lenguajes utilizan las librerías.
- sus colegas, que no tienen relación de parentesco pero están en el mismo paquete;
- una clase Extranjera, sin ninguna relación con la clase Madre.

En el esquema anterior, así como en los siguientes, la flecha simboliza la relación de herencia.

En los párrafos siguientes vamos a examinar sucesivamente los diferentes tipos de protección que Java ofrece a los atributos y a los métodos. Observemos ya desde ahora que hay cinco tipos de protección posibles y que, en todos los casos, la protección va

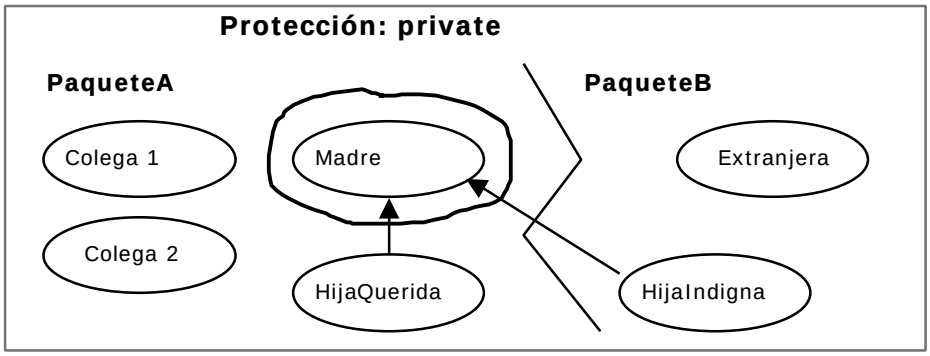
sintácticamente al principio de definición, como en los dos ejemplos siguientes, donde private y public definen los niveles de protección:

```
private void Metodo ();
public int Atributo;
```

Private

La protección más fuerte que puede dar a los atributos o a un método es la protección private.

Esta protección impide a los objetos de otras clases acceder a los atributos o a los métodos de la clase considerada. En el dibujo siguiente, un muro rodea la clase Madre e impide a las otras clases acceder a aquellos de sus atributos o métodos declarados como private.



Insistimos en el hecho de que la protección no se aplica a la clase globalmente, sino a algunos de sus atributos o métodos, según la sintaxis siguiente:

```
private void MetodoMuyProtegido () {
      //     ...
}
private int AtributoMuyProtegido;
public int OtraCosa;
```

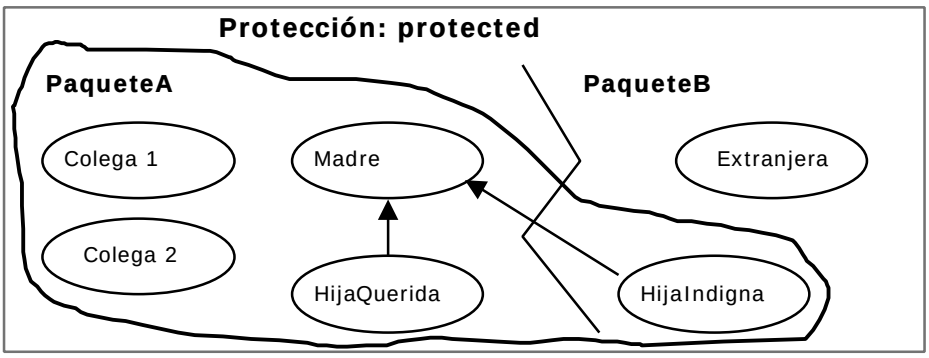
Observe que un objeto que haya surgido de la misma clase puede acceder a los atributos privados, como en el ejemplo siguiente:

```
class Secreta {
      private int s;
      void init (Secreta otra) {
            s = otra.s;
      }

}
```

La protección se aplica pues a las relaciones entre clases y no a las relaciones entre objetos.

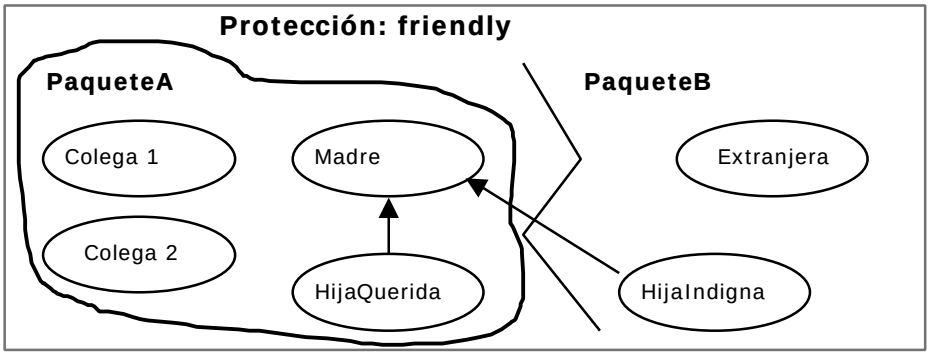
Protected

El tipo de protección siguiente viene definido por la palabra clave protected simplemente. Permite restringir el acceso a las subclases y a las clases del mismo paquete.



Friendly

El tipo de protección predeterminado se llama friendly. Esto significa que si no indica ningún nivel de protección para un atributo o un método, el compilador considera que lo ha declarado friendly. No tiene por qué indicar esta protección explícitamente. Además, friendly no es una palabra clave reconocida. Esta protección autoriza el acceso a las clases del mismo paquete, pero no forzosamente a las subclases.



Los miembros con este tipo de acceso pueden ser accedidos por todas las clases que se encuentren en el paquete donde se encuentre también definida la clase.

Para recordar:

- Friendly es el tipo de protección asumido por defecto.
- Un paquete se define por la palabra reservada package.
- Un paquete puede ser nominado o no.
- Varios archivos pueden pertenecer al mismo paquete.
- Solo se permite una cláusula package por archivo.
- Los archivos que no tienen cláusulas package pertenecen al paquete unnamed.

Public

El último tipo de protección es public. Un atributo o un método calificado así es accesible para todo el mundo.

¿Un caso excepcional? No. Muchas clases son universales y proporcionan servicios al exterior de su paquete: las librerías matemáticas, gráficas, de sistema, etc., están destinadas a ser utilizadas por cualquier clase. Por el contrario, una parte de estas clases está protegida, a fin de garantizar la integridad del objeto.

Se suele decir que los miembros públicos conforman la interfaz de la clase con el usuario.

Separación de la interfaz

¿Cuándo debe utilizarse qué? O en otras palabras: ¿cuáles son los diferentes casos de utilización de los mecanismos de protección? Nos proponemos distinguir principalmente dos casos:

- el atributo o el método pertenece a la *interfaz* de la clase: debe ser public;
- el atributo o la clase pertenece al *cuerpo* de la clase: debe ser protegido. Esta protección es diferente según los casos; en general, la protección más fuerte es aconsejable porque es fácil desproteger un atributo, y es mucho más difícil hacerlo inaccesible si ya se utiliza.

La interfaz de la que hablamos es pues la interfaz conceptual de la clase, es decir los atributos y las signaturas de los métodos (tipo devuelto + nombre + parámetros), directamente utilizados desde el exterior porque corresponden a un servicio prestado.

El cuerpo de la clase es la implementación de dicho servicio.

Es decir, la interfaz de la clase es el *qué* -qué hace la clase-, mientras que su cuerpo es el *cómo* -cómo lo hace.

Por esto conviene hacer pública la interfaz y proteger el cuerpo.

Ahora, veamos un ejemplo en pseudocódigo sobre cómo crear un objeto reloj que demuestre cuáles serían sus funciones públicas y sus funciones privadas:

```
Función inicial o Constructor:
    Reloj negro, hora inicial 12:00am;
Funciones Públicas:
    Apagar
    Encender
    Poner despertador;
Funciones Privadas:
    Mecanismo interno de control
    Mecanismo interno de baterías
    Mecanismo de manecillas
```

Al utilizar uno de estos relojes nos importa su operación no su mecanismo interno, por eso existen funciones públicas o privadas. Las funciones públicas son la interfaz que usaremos. El constructor inicializa al objeto en un estado inicial estable para su operación.

Un ejemplo más, esta vez con una computadora:

```
Función inicial o Constructor:
    Computadora portátil compaq, sistema operativo windows98, encendida
Funciones Públicas:
    Apagado
    Teclado
    Pantalla
    Impresora
    Bocinas
Funciones Privadas:
    Caché del sistema
    Procesador
    Dispositivo de Almacenamiento
    Motherboard
```

Obviamente si se abre (físicamente) la computadora se tendrá acceso a todo, sucede lo mismo si se abre el código fuente del archivo de una clase, se puede modificar todo, pero debe quedar bien claro que la razón por la cual se hace todo lo anterior es para organizar mejor el código, no para impedir el acceso a nadie a ciertas cosas, todo es para mejorar la operación general de una clase ocultando la información que no es necesario que conozcamos y exponiendo la que sí.

ATRIBUTOS DE UNA CLASE

Todas las variables se declaran dentro del cuerpo de la clase o dentro de un método. Las variables declaradas dentro de un método son locales a él; las variables declaradas en el cuerpo de la clase se dice que son *miembros* de ella y son accesibles por todos los métodos de la clase.

Por otra parte, además de los atributos de la propia clase, se puede acceder a todos los atributos de la clase de la cual desciende. Por ejemplo, cualquier clase que descienda de la clase **Polygon** hereda los atributos **npoints, xpoints** e **ypoints**. Esto lo veremos con mayor detalle cuando estudiemos herencia.

Finalmente, los atributos miembros de la clase pueden ser *atributos de clase* o *atributos de instancia;* se dice que son atributos *de clase* si se usa la palabra clave static: en ese caso la variable es única para todas las instancias (objetos) de la clase (ocupa un único lugar en memoria). Si no se usa static, el sistema crea un lugar nuevo para esa variable con cada instancia (o sea que es independiente para cada objeto).

Los atributos pueden ser:

- tipos básicos. (no son clases)
- arreglos *(arrays)*
- clases e interfases (clases de tipos básicos, clases propias, de terceros, etc.)

La lista de atributos sigue el mismo formato de C: se define primero el tipo y luego el nombre del atributo y, finalmente, un ";". Por ejemplo:

```
int n_entero;
float n_real;
char p;
```

La declaración sigue siempre el mismo esquema:

[**Modificador de acceso**] [**static**] [**final**] [**transient**] [**volatile**] Tipo

NombreVariable [**=** Valor];

El modificador de acceso puede ser alguno de los que vimos anteriormente (private, protected, public, etc).

Static y final

static sirve para definir un atributo como de clase, o sea, único para todos los objetos de ella.

En cuanto a **final**, como en las clases, determina que un atributo no pueda ser sobreescrito o redefinido, es decir que no se trata de una variable, sino de una *constante.*

Transient y volatile

Son casos bastante particulares y que no habían sido implementados en Java 1.0.

Transient denomina atributos que no se graban cuando se archiva un objeto, o sea que no forman parte del estado permanente de éste.

Volatile se utiliza con variables modificadas en forma asincrónica por objetos en diferentes *threads* (literalmente "hilos", tareas que se ejecutan en paralelo). Básicamente, esto implica que distintas tareas pueden intentar modificar la variable de manera simultánea, y volatile asegura que se vuelva a leer la variable (por si fue modificada) cada vez que se la va a usar.

Miembros estáticos de una clase

Le hemos explicado anteriormente que cada objeto poseía sus propios atributos. Es posible que todos los objetos de una misma clase tengan atributos en común: son los atributos de clase, introducidos por la palabra clave static. Estos atributos son legibles y modificables por todos los objetos de una misma clase. La modificación de un atributo static es tenida en cuenta inmediatamente por los otros objetos, porque lo *comparten.*

Un miembro de una clase se puede declarar estático (static). Para un miembro dato, la designación static significa que existe sólo una instancia de ese miembro. Un miembro dato estático es compartido por todos los objetos de una clase y existe incluso si ningún objeto de esta clase existe siendo su valor común a la clase completa.

A un miembro dato static se le asigna una zona fija de almacenamiento en tiempo de compilación, al igual que una variable global, pero el identificador de la variable está dentro del ámbito utilizando solamente el operador de resolución con el nombre de la clase.

Ejemplo:

```
import java.io.*

class Participante {
       static int participado = 2;
       int noparticipado = 2;
       void Modifica () {
              participado = 3;
              noparticipado = 3;
       }
}

class demostatic {
       static public void main (String [] arg) {
              Participante p1 = new Participante ();
              Participante p2 = new Participante ();
              System.out.println("pl: " + pl.participado + " " +
                     pl.noparticipado);
              pl.Modifica ();
              System.out.println("pl: " + pl.participado + " " +
                     pl.noparticipado);
              System.out.println("p2: " + p2.participado + " " +
                     p2.noparticipado);
       }
}
```

dará como resultado:

```
C:\Programasjava\objetos>java demostatic
pl: 2 2
pl: 3 3
p2: 3 2
```

En efecto, la llamada a Modifica () ha modificado el atributo participado. Este resultado se extiende a todos los objetos de la misma clase, mientras que sólo ha modificado el atributo noparticipado del objeto actual.

Otro ejemplo:

Sea una clase denominada Alumno con dos miembros dato, la nota de selectividad, y un miembro estático denominado nota de corte. La nota es un atributo que tiene un valor distinto para cada uno de los alumnos u objetos de la clase Alumno, mientras que la nota de corte es un atributo que tiene el mismo valor para a un conjunto de alumnos. Se define también en dicha clase una función miembro que determine si está (true) o no (false) admitido.

```
public class Alumno {
    double nota;
```

```
    static double notaCorte=6.0;
    public Inicializa(double nota) {
        this.nota=nota;
    }
    boolean estaAdmitido(){
        return (nota>=notaCorte);
    }
}
```

En una clase AlumnoApp, utilizamos los objetos de la siguiente forma:

```
import java.io.*
class AlumnoApp {
      static public void main (String [] arg) {

            //Creamos un array de cuatro alumnos y asignamos a cada uno
            //de ellos una nota.
            Alumno[] alumnos={new Alumno(), new Alumno(),
            new Alumno(), new Alumno()};

            //inicializa la nota de los alumnos
            alumnos[0].Inicializa(5.5);
            alumnos[1].Inicializa(6.3);
            alumnos[2].Inicializa(7.2);
            alumnos[3].Inicializa(5.9);

            //Contamos el número de alumnos que están admitidos
            int numAdmitidos=0;
            for(int i=0; i<alumnos.length; i++){
                  if (alumnos[i].estaAdmitido()){
                        numAdmitidos++;
                  }
            }
            System.out.println("admitidos "+numAdmitidos);

            //Accedemos al miembro dato notaCorte desde un objeto de la
            //clase Alumno, para cambiarla a 7.0
            alumnos[1].notaCorte=7.0;

            //Comprobamos que todos los objetos de la clase Alumno
            //tienen dicho miembro dato estático notaCorte cambiado a
            //7.0
            for(int i=0; i<alumnos.length; i++){
                System.out.println("nota de corte "+
                        alumnos[i].notaCorte);
                  }
      }
}
```

El miembro dato notaCorte tiene el modificador static y por tanto está ligado a la clase más que a cada uno de los objetos de dicha clase. Se puede acceder a dicho miembro con la siguiente sintaxis

```
Nombre_de_la_clase.miembro_estático
```

Si el el ejemplo ponemos

```
Alumno.notaCorte=6.5;
for(int i=0; i<alumnos.length; i++){
      System.out.println("nota de corte "+alumnos[i].notaCorte);
}
```

Veremos que todos los objetos de la clase Alumno habrán cambiado el valor del miembro dato estático notaCorte a 6.5.

METODOS DE UNA CLASE

Las clases describen pues los atributos de los objetos, y proporcionan también los métodos. Un método es una función que se ejecuta sobre un objeto. No se puede ejecutar

un método sin precisar el objeto sobre el que se aplica (salvo una excepción que veremos más adelante).

Los métodos de una clase definen las operaciones que un usuario puede realizar con los atributos de la clase. Desde el punto de vista de la POO, el conjunto de todas las funciones definen el conjunto de mensajes a los que los objetos de las citadas clases pueden responder.

Declaración y definición

Los métodos, como las clases, tienen una declaración y un cuerpo. La declaración es del tipo:

[**modificador de acceso**] [**static**] [**abstract**] [**final**] [**native**]

[**synchronized**] TipoDevuelto NombreMétodo

(tipo1 nombre1 [**,** tipo2 nómbre2]...]) [**throws** excepción [**,** excepción2].

La declaración y definición se realizan juntas, es decir en el cuerpo de la clase.

Básicamente, los métodos son como las funciones de C: implementan el cálculo de algún parámetro (que es el que devuelven al método que los llama) a través de funciones, operaciones y estructuras de control. Sólo pueden devolver un valor (del tipo TipoDevuelto), aunque pueden no devolver ninguno (en ese caso TipoDevuelto es void). El valor de retorno se especifica con la instrucción return, dentro del método.

Los métodos pueden utilizar valores que les pasa el objeto que los llama *(parámetros),* indicados con tipo1 nombre1, tipo2 nombre2... en el esquema de la declaración. Estos parámetros pueden ser de cualquiera de los tipos válidos. Si son tipos básicos, el método recibe el *valor* del parámetro; si son *arrays,* clases o interfaces, recibe un puntero a los datos *(referencia).* Veamos un pequeño ejemplo:

```
Public int AumentarCuenta(int cantidad) {
       cnt = cnt + cantidad;
       return cnt;
}
```

Este método, si lo agregamos a la clase Contador, le suma cantidad al acumulador cnt. En detalle:

- el método recibe un valor entero (cantidad).
- lo suma a la variable de instancia cnt.
- devuelve la suma (return cnt).

El **modificador de acceso** puede ser alguno de los que vimos anterioremente (private, protected, public, etc).

El resto de la declaración

Los métodos estáticos (**static**) son, como los atributos, métodos *de clase:* si el método no es static, es un método *de instancia.* El significado es el mismo que para los atributos: un método static es compartido por todas las instancias de la clase.

Los métodos abstractos (**abstract**) son aquellos de los que se da la declaración pero no la implementación (o sea que consiste sólo en el encabezamiento). Cualquier clase que contenga al menos un método abstracto (o cuya clase madre contenga al menos un método abstracto que no esté implementado en la hija) es una clase abstracta.

Un método es final (**final**) cuando no puede ser redefinido por ningún descendiente de la clase.

Los métodos **native** son aquellos que se implementan en otro lenguaje propio de la

máquina (por ejemplo, C o C++). Sun aconseja utilizarlas bajo riesgo propio, ya que, en realidad, son ajenas al lenguaje. Pero existe la posibilidad de usar viejas bibliotecas que uno armó y no tiene ganas de reescribir, ¡a costa de perder portabilidad!

Los métodos **synchronized** permiten sincronizar varios *threads* para el caso en que dos o más accedan concurrentemente a los mismos datos.

Finalmente, la cláusula **throws** sirve para indicar que la clase genera determinadas excepciones.

El cuerpo de los métodos

En Java dentro de los métodos pueden incluirse:

- Declaración de variables locales
- Asignaciones a variables
- Operaciones matemáticas
- Llamados a otros métodos
- Estructuras de control
- Excepciones (try, catch, que veremos más adelante)

Declaración de variables locales

Las variables locales en un método se declaran igual que en C:

Tipo NombreVariable [= Valor];

Por ejemplo:

```
int    suma;
float  precio;
Contador laCuenta;
```

Las variables pueden inicializarse al momento de su creación:

```
int    suma=0;
float  precio = 12.3;
Contador laCuenta = new Contador() ;
```

Llamadas a métodos

Se llama a un método de la misma clase simplemente con el nombre del método y los parámetros entre paréntesis, como se ve, entre otros, en el ejemplo siguiente:

Otra vez recordaremos la clase Contador:

```
//Implementación de un contador sencillo
//GRABAR EN UN ARCHIVO "Contador.java"
// COMPILAR CON: "javac Contador.Java
public class Contador {   //     Se declara y define la clase Contador
       int cnt;
       public void Inicializa() {
              cnt=0;       //inicializa
       }
       //Otros métodos
       public int incCuenta() {
              cnt++;
              return cnt;
       }
       public int getCuenta() {
              return cnt;
       }
}
```

Pueden probar esta clase (mínima) con el siguiente ejemplo de aplicación:

```
// Archivo:  Ejemplollamadas.java
//Compilar con: javac Ejemplollamadas.java
//Ejecutar con: java Ejemplollamadas
import java.io.*;

public class Ejemplollamadas {
       public static void main(String args[]) {
       Contador c = new Contador;
       c.Inicializa();
       System.out.println(c.incCuenta());
       System.out.println(c.getCuenta());
       }
}
```

las líneas en letra resaltada es un ejemplo de un llamado a un método de un objeto de otra clase. Noten que en este caso, es necesario llamar al método para un objeto (instancia) existente, por lo que se indica:

Nombre_del_Objeto<punto>Nombre_del_método(parámetros)

El objeto actual (puntero this)

Nunca se puede llamar una función miembro de una clase a menos que se asocie con un objeto (una instancia de la clase). ¿Cómo sabe una función miembro cuál es la instancia de una clase (el objeto específico) asociada con ella?.

El método utilizado por Java es añadir un argumento extra oculto a las funciones miembro. Este argumento es un puntero al objeto de la clase que los enlaza con la función asociada y recibe un nombre especial denominado **this**.

Dentro de una función miembro, this apunta al objeto asociado con la invocación de la función miembro. Normalmente, el programador no necesita preocuparse por este puntero, ya que el lenguaje realiza la operación automáticamente transparente a las funciones miembro que la utilizan.

Las referencias a los miembros del objeto asociado a la función se realiza con el prefijo this y el operador de acceso punto **.** .

Si tomamos como ejemplo la siguiente clase:

```
public class Cliente {
       private int codigo;
       private float importe;
       public int   getCodigo()   { return codigo; }
       public float getImporte() { return importe; }
       public void setImporte(float x) { importe = x; }
};
```

Cuando escribimos en el método setImporte

```
importe = x;
```

para asignar un valor a importe, no tuvimos necesidad de indicar a qué objeto pertenecía. Cuando no se pone el objeto antes del atributo, se asume que la variable es un miembro del objeto, o es una variable local o parámetro. Si los nombres colisionan, como podría ser en el siguiente método

```
public void setImporte(float importe)
{
       this.importe = importe;
}
```

usamos this para indicar al objeto actual. Esta palabra reservada siempre está apuntando al objeto actual, y puede usarse como en este caso, para resolver una ambigüedad, o puede usarse, por ejemplo, como parámetro en una llamada a una función

para pasar un puntero al objeto asociado:

```
objeto.procesar(this);
```

Métodos especiales

Métodos sobrecargados

Tradicionalmente, un método o una función realizan una tarea específica para la que están programados. Java soporta la *sobrecarga de métodos,* lo que le permite definir versiones de éstos con el mismo nombre en una clase, siempre y cuando las versiones tengan diferentes firmas. Una firma incluye el nombre del método, el número, el orden y los tipos de sus parámetros formales. Como ejemplo simple, considere el reestablecímiento del saldo de cuenta de un objeto Account, esta clase (Account) ya tiene un método

```
public double balanceo()
{
        return saldo;
}
```

que se utiliza para recuperar el saldo de la cuenta. Con la sobrecarga podemos definir un método:

```
public void balanceo(double valor)
{
        saldo = valor;
}
```

que fija el saldo en una cantidad específica. El mismo método recuperará o establecerá el saldo, dependiendo de que se proporcione un argumento; ¡es muy útil!

Tenga en cuenta que, sólo ocurre la sobrecarga, cuando se utiliza varias veces el mismo nombre de método dentro de una clase. No hay un límite práctico para las versiones que es posible apilar en el mismo nombre de método.

Resolución de llamada a un método

Cuando se hace una llamada aun método sobrecargado Java deduce automáticamente, a partir de los argumentos reales, la versión correcta del método que habrá de invocar. A esta actividad se le denomina *resolución de llamada,* Java la realiza al seleccionar un método entre los accesibles *que son aplicables.*

Un método es *aplicable* si toma el mismo número de parámetros que los argumentos dados y cada uno de éstos puede transformarse por *conversión de invocación de método* al tipo del parámetro.

El compilador realiza la resolución de llamada de método al comparar el número y tipo de los argumentos reales con firmas de todos los métodos accesibles y elige un método aplicable que es el *más* especifico.

En resumen, las funciones sobrecargadas tienen el mismo nombre, pero deben tener un número diferente de argumentos o diferentes tipos de argumentos, o ambos. Por ejemplo:

```
void visualizar();
void visualizar(int cuenta);
void visualizar(int cuenta, int max);
```

Ejemplo:

Supongamos que tenemos una clase Media, que calcula la media de dos y tres números, enteros y reales. Para esto tendremos las siguientes funciones:

```
float media (float, float);      //calcula la media de dos valores tipo float
int media (int, int);            //calcula la media de dos valores tipo int
float media (float, float, float); //calcula la media de tres valores tipo float
```

```
int media (int, int, int);       // calcula la media de tres valores tipo float
```

Entonces:

```
public class Media {
       public float Cal_Media (float a, float b)
       { return (a+b)/2.0; }
       public int Cal_Media (int a, int b)
       { return (a+b)/2; }
       public float Cal_Media (float a, float b, float c)
       { return (a+b+c)/3.0;}
       public int Cal_Media (int a, int b, int c)
       { return (a+b+c)/3; }
};

public class demoMedia {
       public static void main (String arg[]) {
              Media M = new Media();
              float x1, x2, x3;
              int y1, y2, y3;
              ...
              System.out.println(M.Cal_Media (x1, x2));
              System.out.println(M.Cal_Media (x1, x2, x3));
              System.out.println(M.Cal_Media (y1, y2));
              System.out.println(M.Cal_Media (y1, y2, y3));
       }
}
```

Métodos constructores

Para cada clase, pueden definirse uno o más métodos particulares: son los constructores.

Un constructor es una función especial que sirve para construir o inicializar objetos.

En general:

- Tienen el mismo nombre de la clase que inicializa.
- No devuelven valores.
- Pueden admitir parámetros como cualquier otra función.
- Pueden existir más de un constructor, e incluso no existir.
- Si no se define ningún constructor de una clase, el compilador generará un constructor por defecto.
- Se llaman en el momento de la creación del objeto.

```
//Implementación de un contador sencillo
//GRABAR EN UN ARCHIVO "Contador.java"
// COMPILAR CON: "javac Contador.Java
public class Contador {   //     Se declara y define la clase Contador
       int cnt;
       public Contador() {
              cnt = 0;            //inicializa en 0
       }
       public Contador(int c) {
              cnt = c;            //inicializa con el valor de c
       }
       //Otros métodos
       public int getCuenta() {
              return cnt;
       }
       public int incCuenta() {
              cnt++;
              return cnt;
       }
```

```
}
```

Pueden probar esta clase (mínima) con el siguiente ejemplo de aplicación:

```
// Archivo:  EjemploConstructor.java
//Compilar con: javac EjemploConstructor.java
//Ejecutar con: java EjemploConstructor
import java.io.*;

public class EjemploConstructor {
       public static void main(String args[]) {
       Contador c1 = new Contador;
       Contador c2 = new Contador(20);
       System.out.println(c1.getCuenta());
       System.out.println(c2.getCuenta());
       }
}
```

Cuando, desde una aplicación u otro objeto, se crea una instancia de la clase Contador, mediante la instrucción:

```
Contador c1 = New Contador();
```

el compilador busca un método con el mismo nombre de la clase y que se corresponda con la llamada en cuanto al tipo y número de parámetros. Dicho método se llama Constructor, y una clase puede tener más de un constructor (no así un objeto instanciado, ya que, una vez que fue creado, **no** puede "recrearse" a sí mismo).

En tiempo de compilación o ejecución, según corresponda, al encontrarse dicha instrucción, se reserva espacio para el objeto instanciado y se crea su estructura y en tiempo de ejecución se llama al método constructor.

La utilización de las instrucciones vistas, por ejemplo new(), entraña en efecto la creación física del objeto y la llamada a uno de sus constructores. Si hay varios constructores, estos difieren unos de otros por los parámetros que se les pasa mediante new().

Por ejemplo:

```
import java.io.*;
// una clase que tiene dos constructores diferentes
class Ejemplo {
       public Ejemplo (int param) {
              System.out.println ("Ha llamado al constructor");
              System.out.println ("con un parámetro entero");
       }
       public Ejemplo (String param) {
              System.out.println ("Ha llamado al constructor'.);
              System.out.println ("con un parámetro String");
       }
}

// una clase que sirve de main
public class democonstructor {
       public static void main (String arg[]) {
              Ejemplo e;
              e = new Ejemplo (2);
              e = new Ejemplo ("2");
       }
}
```

da el resultado siguiente:

```
c:\Programasjava\objetos>java democonstructor
Ha llamado al constructor con un parámetro entero
Ha llamado al constructor con un parámetro String
```

Los constructores no tienen tipo de retorno. Atención, si por error definimos un constructor que tenga un tipo de retomo, el compilador lo considerará como un método normal. En ese caso tendremos la impresión de que el constructor no se llama en el momento de la creación del objeto. En realidad, se llamará a un constructor predeterminado, porque no habremos definido realmente un constructor.

Tipos de constructores

Constructores por defecto

El constructor por defecto es un constructor que no acepta argumentos. Por ejemplo:

```
class Punto {
        int x;
        int y;
        public Punto()
        {       x = 0;
                y = 0;
        }
}
```

Para crear objetos usando este constructor se escribiría:

```
public class constructorPorDefecto {
        public static void main (String arg[]) {
                Punto p1;
                p1 = new Punto();
        }
}
```

Constructores con argumentos

El constructor con argumentos, como su nombre lo indica posee argumentos. Por ejemplo:

```
class Punto {
        int x;
        int y;
        public Punto(int x, int y)
        {       this.x = x;
                this.y = y;
        }
}
```

Para crear objetos usando este constructor se escribiría:

```
public class constructorArgumentos {
        public static void main (String arg[]) {
                Punto p2;
                p2 = new Punto(2, 4);
        }
}
```

Constructores copiadores

Un constructor que crea un objeto a partir de uno existente se llama constructor copiador o de copias. Es un constructor que toma como único parámetro otro objeto del mismo tipo. El constructor de copia tiene sólo un argumento: una referencia a un objeto de la misma clase. Por ejemplo:

```
class Punto {
        int x;
        int y;
        public Punto(Punto p)
        {       x = p.x;
```

```
                y = p.y;
        }
}
```

Para crear objetos usando este constructor se escribiría:

```
public class constructorCopia {
        public static void main (String arg[]) {
                Punto p2;                   //del ejemplo anterior
                p2 = new Punto(2, 4);

                Punto p3;
                p3 = new Punto(p2);         //p2 sería el objeto creado en el
                                            //ejemplo anterior
        }
}
```

o bien

```
//p2 sería el objeto creado en el ejemplo anterior
public class constructorCopia {
        public static void main (String arg[]) {
                Punto p2;                   //del ejemplo anterior
                p2 = new Punto(2, 4);

                Punto p3;
                p3 = p2;            //p2 sería el objeto creado en el
                                    //ejemplo anterior
        }
}
```

En esta asignación no se crea ningún objeto, solamente se hace que p2 y p3 apunten y referencien al mismo objeto.

Caso especial

En Java es posible inicializar los atributos indicando los valores que se les darán en la creación del objeto. Estos valores se adjudican tras la creación física del objeto, pero antes de la llamada al constructor.

```
import java.io.*
class Reserva {
        int capacidad = 2;
        // valor predeterminado
        public Reserva (int lacapacidad) {
                System.out.println (capacidad);
                capacidad = lacapacidad;
                System.out.println (capacidad);
        }
}

class demovalor {
        static public void main (String []arg) {
                new Reserva (1000);
        }
}
```

visualizará sucesivamente el valor que el núcleo ha dado al atributo capacidad tras la inicialización (2) y posteriormente el valor que le asigna el constructor (1000).

Añadamos que los atributos no inicializados explícitamente por el desarrollador tienen valores predeterminados, iguales a cero. Así, si no se inicializa capacidad:

```
class Reserva {
        int capacidad:
        public Reserva (int lacapacidad) {
                System.out.println (capacidad);
                capacidad = lacapacidad;
```

```
                    System.out.println (capacidad);
            }
        }
```

el programa indicará los valores 0 seguido de 1000.

HERENCIA

INTRODUCCION

La herencia es una propiedad esencial de la Programación Orientada a Objetos que consiste en la creación de nuevas clases a partir de otras ya existentes.

Java permite heredar a las clases características y conductas de una o varias clases denominadas base. Las clases que heredan de clases base se denominan derivadas, estas a su vez pueden ser clases bases para otras clases derivadas. Se establece así una clasificación jerárquica, similar a la existente en Biología con los animales y las plantas.

La herencia ofrece una ventaja importante, permite la reutilización del código. Una vez que una clase ha sido depurada y probada, el código fuente de dicha clase no necesita modificarse. Su funcionalidad se puede cambiar derivando una nueva clase que herede la funcionalidad de la clase base y le añada otros comportamientos. Reutilizando el código existente, el programador ahorra tiempo y dinero, ya que solamente tiene que verificar la nueva conducta que proporciona la clase derivada.

Los programadores crean clases base:

- Cuando se dan cuenta que diversos tipos tienen algo en común, por ejemplo en el juego del ajedrez peones, alfiles, rey, reina, caballos y torres, son piezas del juego. Creamos, por tanto, una clase base y derivamos cada pieza individual a partir de dicha clase base.
- Cuando se precisa ampliar la funcionalidad de un programa sin tener que modificar el código existente.

En el lenguaje Java, todas las clases derivan implícitamente de la clase base Object, por lo que heredan las funciones miembro definidas en dicha clase. Las clases derivadas pueden redefinir algunas de estas funciones miembro como toString y definir otras nuevas.

Heurísticas para crear subclases

La regla más importante para saber cuando una clase debe convertirse en subclase de otra o cuándo es más apropiado otro mecanismo es que, para que una clase se relacione con otra por medio de la herencia, debe haber una relación de funcionalidad entre ambas. Entonces describimos la regla *es-un* para captar la relación.

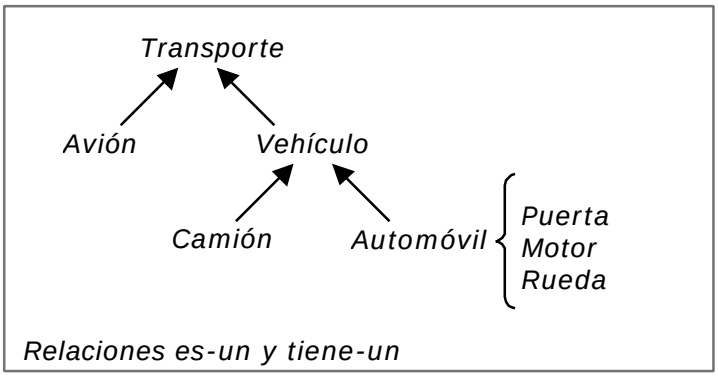
Relaciones es-un y tiene-un

Hay dos relaciones de importancia conocidas coloquialmente como la relación *es-un* y la relación *tiene-un* (o es *parte-de*).

Se da la relación *es-un* entre dos conceptos cuando el primero es un ejemplar especificado del segundo. Es decir, para todo propósito práctico, el comportamiento y los datos asociados con la idea más específica forman un subconjunto del comportamiento y los datos asociados con la idea más abstracta. Por ejemplo, un florista es-un comerciante, puesto que los hechos generales de los comerciantes son aplicables a los floristas.

Por otra parte, la relación tiene-un se da cuando el segundo concepto es un componente del primero, pero cuando los dos no son, en sentido alguno, la misma cosa, sin

importar cuán abstracta sea la generalidad. Por ejemplo, un automóvil tiene-un motor, aunque claramente es el caso de que un automóvil no es-un motor ni un motor es-un automóvil. Sin embargo un automóvil es-un vehículo, el cual es a su vez es-un transporte como lo muestra el gráfico siguiente:

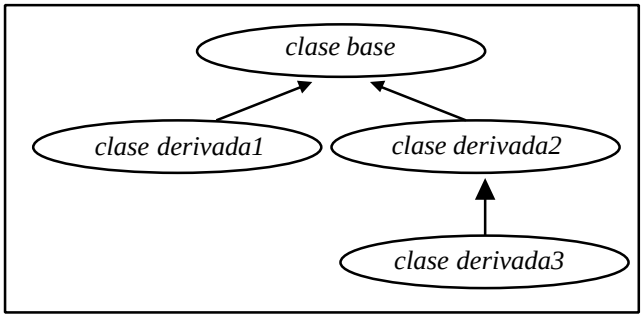


Al decir que una subclase es una superclase, estamos diciendo que la funcionalidad y los datos que asociamos con la clase hija (o subclase) forman un superconjunto de la funcionalidad y los datos asociados con la clase paterna (o superclase).

HERENCIA SIMPLE

Como ya dijimos la *herencia simple* es aquella en la que cada clase derivada hereda de una única clase base.

Un diagrama representativo de herencia simple puede ser el siguiente:



En este ejemplo se muestran tres derivaciones simples de clases, la clase derivada1 y la clase derivada2 derivan de la clase llamada base y la clase derivada3 deriva de la clase llamada derivada2.

En general para obtener herencia se necesitan como mínimo dos clases, la clase base y la clase derivada.

Clases bases

Son clases que sirven de modelos para otras, no pueden tener el especificador final ya que una clase final es aquella que no puede tener clases que la hereden.

Por ejemplo:

```
class clase_base
{ ... };
```

Clases derivadas

Las clases heredadas se llaman clases derivadas, clases hijas o subclases.

Las características de las clases derivadas son:

- Puede a su vez ser una clase base, dando lugar a una jerarquía de clases.
- Hereda todos los miembros de la clase base, pero solo podrá acceder a aquellos que los especificadores de acceso de la clase base lo permitan.
- Puede añadir a los miembros heredados, sus propios atributos y métodos.
- Los miembros heredados por una clase derivada, pueden a su vez ser heredados por más clases derivadas de ella.
- Las clases derivadas solo pueden acceder a los miembros públicos (public), protegidos (protected) y protegidos-privados (private protected) de la clase base o clases bases, como si fueran miembros de ella misma.
- No tiene acceso a los miembros privados de la clase base.
- Los siguientes elementos de la clase base no se heredan:
    - Constructores
    - Funciones y datos estáticos de la clase

Declaración de una clase derivada

La declaración de una clase derivada tiene la siguiente sintaxis:

```
class clase_derivada extends clase_base
{ ... };
```

La instrucción extends indica de qué clase desciende la nuestra. Si se omite, Java asume que desciende de la superclase **object**.

Cuando una clase desciende de otra, significa que hereda sus atributos y sus métodos. Esto quiere decir que, a menos que los redefinamos, sus métodos serán los mismos que los de la clase madre y podrán utilizarse en forma transparente, siempre y cuando no sean *privados* en la clase madre, o protegidos o propios del paquete para subclases de otros paquetes.

Por ejemplo:

```
class base {
      protected int enteroCualquiera;
      public    void Metodo ()  { ;};
}

class derivada extends base {           //extends por "hereda"
      void MetodoPropioDeHija () {
            //     ...
      }
}
```

Otro ejemplo:

```
class Vehiculo {
      protected int numRuedas;
      protected int velocidad;
      public    void Acelerar() {
            velocidad += 100; }
};

class Camion extends Vehiculo {
      int pesocarga;      // un nuevo atributo específico de Camion
};

class Coche extends Vehiculo {
      boolean pinturaMetalizada;       //un nuevo atributo específico de
                                       //coche
};
```

```
public class demoHerencia {
      public static void main (String arg[]) {
            // distintas formas de crear clases derivadass

            Coche unVehiculo1 = new Coche();
            unVehiculo1.Acelerar();

            Camion unVehiculo2 = new Camion();
            unVehiculo2.Acelerar();


            Vehiculo unVehiculo3 = new Coche();
            unVehiculo3.Acelerar();

            unVehiculo3 = new Camion();
            unVehiculo3.Acelerar();
      }
}
```

Si Hija es una clase que hereda de Madre, entonces todo objeto del tipo Hija es automáticamente del tipo Madre.

Atributos y Métodos

Las clases derivadas retoman los atributos y los métodos de la clase base. Pero puede:

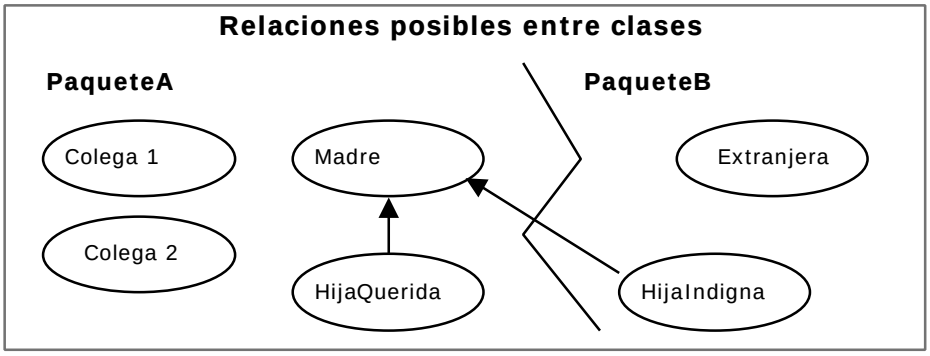
- enriquecerlos con nuevos atributos y nuevos métodos;
- redefinir los métodos.

Acceso a miembros de clases bases

El acceso, de la clase derivada, a los miembros de la clase base está regulado por los especificadores de acceso vistos en la unidad anterior.

Para analizar el acceso a los miembros en herencia, presentaremos seguidamente un ejemplo que retoma las clases siguientes:

- Madre, que define atributos y métodos más o menos protegidos;
- HijaQuerida, que hereda de ella y que se encuentra en el mismo paquete;
- HijaIndigna, que también hereda de Madre pero que está en otro paquete;
- Colega, que no hereda de Madre, pero que está en el mismo paquete.



Las tres últimas clases intentan acceder a los atributos de Madre o utilizar sus métodos. No hemos utilizado Extranjera, sin vínculos de familia con Madre, y que está en

otro paquete, porque sólo los atributos y los métodos public de Madre le son accesibles.

La clase Madre tiene cuatro atributos y cuatro métodos, correspondientes a los diferentes niveles de protección. No hemos utilizado public, para no cargar aún más estos ejemplos. He aquí la clase Madre:

```
package PaqueteA {
//     significa que las clases pertenecen al PaqueteA
       class Madre {
              private int atrprivate = 0;
              private protected int atrprivateprotected = 0;
              protected int atrprotected = 0;
              int atrFriendly = 0;

              private void metprivate () {...}
              private protected void metprivateprotected () {...}
              protected void metprotected () {...}
              void metFriendly () {...}
```

He aquí la clase HijaQuerida. Las líneas en comentario son lineas incorrectas. Observará que los atributos y métodos private protected de Madre no pueden utilizarse más que si se refieren a un objeto de la clase HijaQuerida; por el contrario, está prohibido acceder a ellos si se refieren a un objeto de la clase Madre.

```
class HijaQuerida extends Madre {
       void AccesAMadre (Madre m) {
              //m.atrprivate = 0;                    //prohibido
              //m.atrprivateprotected = 0;           //prohibido
              m.atrprotected = 0;
              m.atrFriendly = 0;
              //m.metprivate () {...}                //prohibido
              //m.metprivateprotected () {...}       //prohibido
              m.metprotected () {...}
              m.metFriendly (){...}
       }
       void AccessAHijaQuerida () {
              //atrprivate = 0;                      //prohibido
              atrprivateprotected = 0;
              atrprotected = 0;
              atrFriendly = 0;
              //II   metprivate () {...}             //prohibido
              metprivateprotected () {...}
              metprotected (){...}
              metFriendly (){...}
       }
};
```

La clase HijaIndigna tiene menos derechos:

```
import PaqueteA.Madre;
class HijaIndigna extends Madre {       // extends significa hereda de
       void AccessAMadre (Madre m) {
              //m.atrPrivate = 0;                    //prohibido
              //m.atrPrivateprotected = 0;           // prohibido
              //m.atrProtected = 0;                  // prohibido
              //m.atrFriendly = 0;                   // prohibido
              //m.metPrivate () {...}                // prohibido
              //m.metprivateprotected () {...}       // prohibido
               //m.metprotected () {...}             // prohibido
              //m.metFriendly () {...}               // prohibido
       }
       void AccessAHijaIndigna () {
              //atrprivate = 0;                      //prohibido
              atrprivateprotected = 0;
              atrProtected = 0;
              //atrFriendly = 0;                     //prohibido
              //metprivate (){...}                   //prohibido
```

```
                    metprivateprotected (){...}
                    metProtected (){...}
                    //metFriendly {...}                    //prohibido
```

Finalmente, para terminar, la clase Colega situada en el mismo paquete que la clase Madre:

```
package PaqueteA;

class colega {
      void AccessAMadre (Madre m) {
            //m.atrPrivate = O;                    //prohibido
            //m.atrprivateprotected = 0;           //prohibido
            m.atrprotected = 0;
            m.atrFriendly = 0;
            //m.metPrivate (){...}                 //prohibido
            //m.metprivateprotected (){...}        //prohibido
            m.metprotected (){...}
            m.metFriendly (){...}
```

Para mayor claridad, hemos agrupado en una tabla los diferentes accesos posibles, que son tres:

- S: acceso posible;
- N: acceso imposible;
- R: caso reservado a las subclases. Acceso posible, a condición de que el atributo o el método se refiera a un objeto de la subclase, y no a un objeto de la clase Madre.

Esta tabla es válida tanto para los atributos como para los métodos.

Acceso a partir de:	**private**	**Private protected**	**Protected**	**Friendly**	**public**
Madre	S	S	S	S	S
HijaQuerida	N	R	S	S	S
Colega	N	N	S	S	S
HijaIndigna	N	R	R	N	S
Extranjera	N	N	N	N	S

Constructores en clases derivadas

Cuando se construye una clase derivada se llama a su constructor. Pero sabemos que una clase derivada se construye a partir de una clase base, por lo tanto primero se debería ejecutar el constructor de la clase base. Como los constructores no se heredan, es necesario que el constructor de la clase base sea invocado desde la clase derivada. Esta invocación es implícita o explícita según la presencia o no de constructores explícitos.

Cuando una clase se deriva de otra, el orden de construcción es el siguiente:

1. Constructores de la clase base.
2. Constructores de los miembros de la clase derivada.
3. Ejecución de las instrucciones contenidas en el cuerpo del constructor de la clase derivada.

Si se llama a un constructor de una clase derivada, un constructor de su clase base

debe ser siempre llamado igualmente. Si la llamada no especifica ningún constructor de la clase base, entonces se llamará al constructor por defecto de la clase base.

En resumen, cuando una clase base define un constructor, éste se debe llamar durante la creación de cada instancia de una clase derivada, para garantizar la buena inicialización de los datos miembros que la clase derivada hereda de la clase base. En este caso, la clase derivada debe definir a su vez un constructor que llama al constructor de la clase base proporcionándole los argumentos requeridos:

En la definición del constructor de la clase derivada se utiliza la palabra clave **super** para simbolizar la llamada al constructor de Madre. Muy lógicamente esta palabra clave se sitúa en la primera línea del constructor de Hija.

Los argumentos que llevará la función super se corresponderán con los definidos en los constructores de la clase base, es decir, usamos super() si el constructor de la clase base no recibe argumentos y super(argumentos...) si el constructor de la clase base recibe argumentos.

Por ejemplo:

```
class Mamifero {
  String especie, color;
  public Mamifero() {
     especie = "";
     color = "";
  }
  public Mamifero( String especie, String color) {
     this.especie = especie;
     this.color = color;
  }
  public void mover() {
    System.out.println("El mamífero se mueve");
  }
}

class Gato extends Mamifero {
  int numero_patas;
  public Gato() {
    super();
    numero_patas = 0;
  }
  public Gato( String especie, String color, int numero_patas) {
    super(especie, color);
    this.numero_patas = numero_patas;
  }
  public void mover() {
    super.mover();        //llama al método mover de la clase base
    System.out.println("El gato es el que se mueve");
  }
}

public class EjemploConstructor {
  public static void main(String[] args) {
    Gato bisho = new Gato();
    bisho.mover();
  }
}
```

La palabra clave super

La palabra super no solo sirve para llamar al constructor de la clase base, también sirve para acceder a sus variables y a sus funciones, generalmente se utiliza para llamar funciones y variables de la clase base cuando la clase derivada y la clase base tienen funciones y/o variables con el mismo nombre (de modo que las de la clase derivada ocultan a las de la clase base), es una forma de distinguirlas muy similar al this, por ejemplo el método mover del ejemplo anterior existe tanto en la clase derivada como en la clase base

pero se utilizo super en la función mover de la clase derivada para distinguir explícitamente que se está haciendo referencia a la función mover de la clase base.

REDEFINICION DE METODOS

Cuando se hace heredar una clase de otra, se pueden redefinir ciertos métodos, a fin de refinarlos, o bien de modificarlos. El método lleva el mismo nombre y la misma signatura, pero sólo se aplica a los objetos de la subclase o a sus descendientes.

Así, el programa siguiente:

```
import java.io.*;

class Madre {
	void habla () {
		System.out.println (" Soy de la clase Madre");
	}
};

class Hija extends Madre {
	void habla () {
		System.out.println ("Soy de la clase Hija");
	}
};

class Nieta extends Hija {
	void NuevoMetodoSinHistoria () {
			System.out.println ("No es llamado");
		}
	};

public class redefinicion {
	public static void main (Strinq arg[]) {
		Madre m = new Madre();
		m.habla();
		Hija h = new Hija();
		h.habla ();
		Nieta n = new Nieta ();
		n.habla ();
	}
};
```

dará el resultado siguiente:

```
C: \programasjava\objetos>java redefinicion
Soy de la clase Madre
Soy de la clase Hija
Soy de la clase Hija
```

Los objetos de la clase Madre utilizan el método de la clase Madre, los de la clase Hija el método redefinido en la clase Hija y los de la clase Nieta utilizan el método de la clase Hija, porque este método no ha sido redefinido en la clase Nieta.

Redefinicíón y sobrecarga

Hemos visto que una clase que hereda de otra podía redefinir ciertos métodos. Hemos visto también que se podía dar el mismo nombre a dos métodos diferentes, a poco que tengan parámetros diferentes. ¿Se trata del mismo mecanismo?

¡En absoluto! La sobrecarga permite distinguir dos métodos de la misma clase que pueden ser llamados uno u otro sobre el mismo objeto, pero que poseen parámetros diferentes. La redefinición distingue dos métodos de dos clases de las cuales una es ancestro de la otra y que tienen ambos los mismos parámetros.

Por ejemplo:

```
clase Madre {
      public void metodo ()
      { ... }
      public void metodo (int param) //el método está sobrecargado
      { ... }
};

class Hija extends Madre {
      public void metodo ()     //redefinición de metodo() de Madre
      {...}
};
```

Caso especial (especificador Final)

Si se diseña una clase algo sofisticada que asegura servicios potentes, se puede correr el riesgo de desmoronarse si cualquiera puede heredar de ella y redefinir cualquier método.

Asimismo, Java ha previsto un mecanismo para impedir la redefinición de métodos. Es el empleo de la palabra clave final, que indica al compilador que está prohibido redefinir un método.

Así, el programa siguiente:

```
clase Madre {
      final void Metodosensible () { ...}
};

clase Hija extends Madre {
      void MetodoSensible () {...}
};
```

es rechazado por el compilador:

```
c:\programesjava\objetos>javac final.java
final.java:8: Final methods can't be overriden.
Method void MetodoSensible() is final in class Madre.
void MetodoSensible () {

1 error
```

Los atributos también pueden ser final, lo que permite de hecho definir constantes. En efecto, si un atributo es final:

- debe ser inicializado en su declaración:

```
final int constante;                  // incorrecto
final int constante = 287;            //correcto
```

- no puede ser modificado ni por un método de una subclase, ni por un método de la propia clase.

Si un atributo es final static, es constante y accesible simplemente dando el nombre de la clase. Es pues una verdadera constante.

Ejemplo:

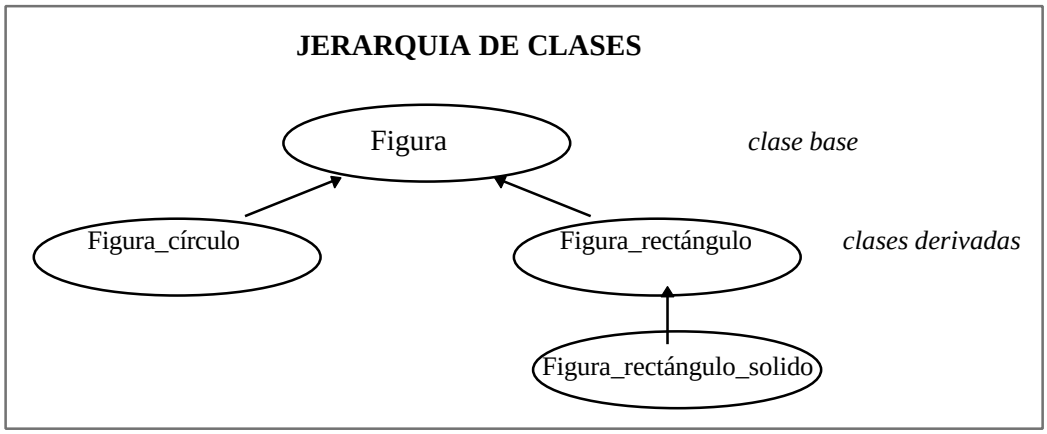
```
double d = Math.E;              // 2.7182818284590452354
d = Math.PI ;                   // 3.14159265358979323846
```

JERARQUIA DE CLASES

Java utiliza un sistema de herencia jerárquica. Es decir se hereda una clase de otra, creando nuevas clases a partir de clases ya existentes. Solo se pueden heredar clases, no funciones ordinarias ni variables.

La relación de una clase base y una clase derivada supone un orden de jerarquía

simple. A su vez, una clase derivada puede ser utilizada como una clase base para derivar más clases. Por consiguiente se puede construir jerarquías de clases, en las que cada clase sirve como padre o raíz de una nueva clase. La siguiente figura representa un diagrama de jerarquía de clases.



Un ejemplo de jerarquía de clases

Vamos a poner un ejemplo que simule la utilización de librerías de clases para crear un interfaz gráfico de usuario como Windows 3.1 o Windows 95.

La clase base

Supongamos que tenemos una clase que describe la conducta de una ventana muy simple, aquella que no dispone de título en la parte superior, por tanto no puede desplazarse, pero si cambiar de tamaño actuando con el ratón en los bordes derecho e inferior.

La clase Ventana tendrá los siguientes miembros dato: la posición x e y de la ventana, de su esquina superior izquierda y las dimensiones de la ventana: ancho y alto.

```
public class Ventana {
    protected int x;
    protected int y;
    protected int ancho;
    protected int alto;
    public Ventana(int x, int y, int ancho, int alto) {
        this.x=x;
        this.y=y;
        this.ancho=ancho;
        this.alto=alto;
    }
//...
}
```

Las funciones miembros, además del constructor serán las siguientes: la función mostrar que simula una ventana en un entorno gráfico, aquí solamente nos muestra la posición y las dimensiones de la ventana.

```
    public void mostrar(){
        System.out.println("posición   : x="+x+", y="+y);
        System.out.println("dimensiones : w="+ancho+", h="+alto);
    }
```

La función cambiarDimensiones que simula el cambio en la anchura y altura de la ventana.

```
    public void cambiarDimensiones(int dw, int dh){
```

```
            ancho+=dw;
            alto+=dh;
        }
```

El código completo de la clase base Ventana, es el siguiente

```
package ventana;

public class Ventana {
    protected int x;
    protected int y;
    protected int ancho;
    protected int alto;
    public Ventana(int x, int y, int ancho, int alto) {
        this.x=x;
        this.y=y;
        this.ancho=ancho;
        this.alto=alto;
    }
    public void mostrar(){
        System.out.println("posición   : x="+x+", y="+y);
        System.out.println("dimensiones : w="+ancho+", h="+alto);
    }
    public void cambiarDimensiones(int dw, int dh){
        ancho+=dw;
        alto+=dh;
    }
}
```

Objetos de la clase base

Como vemos en el código, el constructor de la clase base inicializa los cuatro miembros dato. Llamamos al constructor creando un objeto de la clase Ventana

```
Ventana ventana=new Ventana(0, 0, 20, 30);
```

Desde el objeto ventana podemos llamar a las funciones miembro públicas

```
ventana.mostrar();
ventana.cambiarDimensiones(10, 10);
ventana.mostrar();
```

La clase derivada

Incrementamos la funcionalidad de la clase Ventana definiendo una clase derivada denominada VentanaTitulo. Los objetos de dicha clase tendrán todas las características de los objetos de la clase base, pero además tendrán un título, y se podran desplazar (se simula el desplazamiento de una ventana con el ratón).

La clase derivada heredará los miembros dato de la clase base y las funciones miembro, y tendrá un miembro dato más, el título de la ventana.

```
public class VentanaTitulo extends Ventana{
    protected String titulo;
    public VentanaTitulo(int x, int y, int w, int h, String nombre) {
        super(x, y, w, h);
        titulo=nombre;
    }
```

como ya vimos, extends es la palabra reservada que indica que la clase VentanaTitulo deriva, o es una subclase, de la clase Ventana.

La primera sentencia del constructor de la clase derivada es una llamada al constructor de la clase base mediante la palabra reservada super. La llamada

```
super(x, y, w, h);
```

inicializa los cuatro miembros dato de la clase base Ventana: x, y, ancho, alto. A

continuación, se inicializa los miembros dato de la clase derivada, y se realizan las tareas de inicialización que sean necesarias.

La función miembro denominada desplazar cambia la posición de la ventana, añadiéndoles el desplazamiento.

```
public void desplazar(int dx, int dy){
    x+=dx;
    y+=dy;
}
```

Redefine la función miembro mostrar para mostrar una ventana con un título.

```
public void mostrar(){
    super.mostrar();
    System.out.println("titulo     : "+titulo);
}
```

En la clase derivada se define una función que tiene el mismo nombre y los mismos parámetros que la de la clase base. Se dice que redefinimos la función mostrar en la clase derivada. La función miembro mostrar de la clase derivada VentanaTitulo hace una llamada a la función mostrar de la clase base Ventana, mediante

```
super.mostrar();
```

De este modo aprovechamos el código ya escrito, y le añadimos el código que describe la nueva funcionalidad de la ventana por ejemplo, que muestre el título.

Si nos olvidamos de poner la palabra reservada super llamando a la función mostrar, tendríamos una función recursiva. La función mostrar llamaría a mostrar indefinidamente.

```
public void mostrar(){ //¡ojo!, función recursiva
     System.out.println("titulo     : "+titulo);
     mostrar();
}
```

La definición de la clase derivada VentanaTitulo, será la siguiente.

```
Package ventana;

Public class VentanaTitulo extends Ventana{
    Protected String titulo;
    Public VentanaTitulo(int x, int y, int w, int h, String nombre) {
        super(x, y, w, h);
        titulo=nombre;
    }
    public void mostrar(){
        super.mostrar();
        System.out.println("titulo     : "+titulo);
    }
    public void desplazar(int dx, int dy){
        x+=dx;
        y+=dy;
    }
}
```

Objetos de la clase derivada

Creamos un objeto ventana de la clase derivada VentanaTitulo

```
VentanaTitulo ventana=new VentanaTitulo(0, 0, 20, 30, "Principal");
```

Mostramos la ventana con su título, llamando a la función mostrar, redefinida en la clase derivada

```
ventana.mostrar();
```

Desde el objeto ventana de la clase derivada llamamos a las funciones miembro definidas en dicha clase

```
ventana.desplazar(4, 3);
```

Desde el objeto ventana de la clase derivada podemos llamar a las funciones miembro definidas en la clase base.

```
ventana.cambiarDimensiones(10, -5);
```

Para mostrar la nueva ventana desplazada y cambiada de tamaño escribimos

```
ventana.mostrar();
```

POLIMORFISMO

INTRODUCCION

Una de las características de más importancia de la programación orientada a objetos es la capacidad de que diferentes objetos responden a órdenes similares de modo diferente. Esta característica se denomina polimorfismo y los objetos que lo soportan se llaman objetos polimórficos.

El polimorfismo consiste en que toda referencia a un objeto de una clase específica puede tomar la forma de una referencia a un objeto de una clase heredada a la suya.

El comportamiento polimórfico en Java indica que ciertas variables del código pueden comportarse de distintas maneras, es decir en un momento se comportan como si fueran de cierto tipo y si así lo deseamos pueden tomar otro comportamiento como si fueran una variable distinta.

La clave del polimorfismo en Java esta en la elección de métodos de forma dinámica esto significa que el intérprete de java elige la función apropiada en tiempo de ejecución.

Por ejemplo, supongan que tenemos una impresora por un lado y una televisión por el otro (en términos de objetos de la vida real), estos dos objetos son similares pero no son iguales, aún así se encienden casi de la misma manera: con un botoncito, si alguien te pide que apagues la TV accionas su botoncito y sucede lo mismo con la impresora pero si te piden que mandes a imprimir algo a la televisión pues evidentemente no será posible!!. Bueno pues eso es la elección dinámica de métodos, el intérprete Java realiza esta revisión en tiempo de ejecución (solo que esta vez revisa los archivos de clases), ¿en qué criterio se basa esta elección? se basa en la clase (u objeto) con el cual se esté trabajando actualmente.

A continuación veremos algunos conceptos que nos servirán para implementar polimorfismo en nuestros programas.

METODOS Y CLASES ABSTRACTAS

Como vimos anteriormente, es posible que con la herencia terminemos creando una familia de clases con una interfaz común. En esos casos es posible, y hasta probable, que la clase raíz de las demás no sea una clase útil, y que hasta deseemos que el usuario nunca haga instancias de ella, porque su utilidad es inexistente. No queremos implementar sus métodos, sólo declararlos para crear una interfaz común. Entonces declaramos sus métodos como abstractos:

```
public abstract void mi_metodo();
```

Como vemos, estamos declarando el método pero no implementándolo, ya que sustituimos el código que debería ir entre llaves por un punto y coma. Cuando existe un método abstracto deberemos declarar la clase abstracta o el compilador nos dará un error. Al declarar como abstracta una clase nos aseguramos de que el usuario no pueda crear instancias de ella:

```
abstract class Mamifero {
  String especie, color;
  public abstract void mover();
}
```

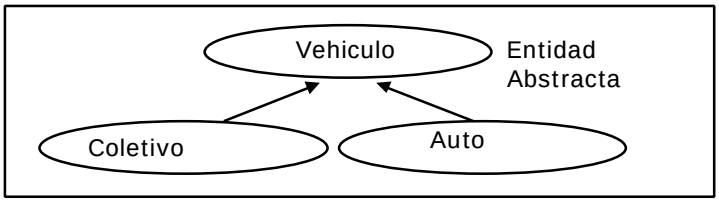
```
class Gato extends Mamifero {
  int numero_patas;
  public void mover() {
    System.out.println("El gato es el que se mueve");
  }
}

public class Abstractos {
  public static void main(String[] args) {
    Gato bisho = new Gato();
    bisho.mover();
  }
}
```

En nuestro ejemplo de herencia, parece absurdo pensar que vayamos a crear instancias de Mamífero, sino de alguna de sus clases derivadas. Por eso decidimos declararlo abstracto.

Las clases abstractas sirven para definir conceptos incompletos, que deben ser completados en las subclases de la clase abstracta.

Por ejemplo en el juego del ajedrez podemos definir una clase base denominada Pieza, con las características comunes a todas las piezas, como es su posición en el tablero, y derivar de ella las características específicas de cada pieza particular. Así pues, la clase Pieza será una clase abstracta con una función abstract denominada mover, y cada tipo de pieza definirá dicha función de acuerdo a las reglas de su movimiento sobre el tablero.

En otro ejemplo, el siguiente gráfico nos muestra una generalización de clases



Así, la clase Vehiculo puede definirse como sigue:

```
abstract class Vehiculo {
       int numpasajeros;
       int Peso;
       abstract void Arrancar ();
       abstract void Correr ();
       void TransportarPasajero (int num) {
              numpasajeros = num;
              Arrancar ();
              Correr ();
       }
};
```

Se capitalizan en esta clase los conocimientos comunes a sus subclases Auto y Colectivo. En ambos casos, hay pasajeros a embarcar antes de Arrancar() y de Correr(). Por el contrario, Arrancar() y Correr() son diferentes en ambas subclases.

Una clase abstracta no puede ser instanciada porque no está completamente definida.

Si las subclases Auto y Colectivo no dan un cuerpo a todos los métodos abstractos, deben a su vez declararse abstractas y no pueden utilizarse directamente.

Interfaces

Las interfaces tienen como misión llevar el concepto de clase abstracta un poco más lejos. Una interface es como una clase abstracta pero no permite que ninguno de sus métodos esté implementado.

En una interface, todos sus métodos son abstractos, y todos sus atributos son final (es decir, no pueden modificarse). Se declaran sustituyendo class por interface:

```
interface ClaseInterface {
      int metodoAbstracto();
      int atributoFinal;
};
```

Como se puede ver, no es necesario especificar que los métodos son abstractos y los atributos son *finales* para este tipo de clases, utilizando abstract y final, dado que estas características van implícitas en la declaración de la clase como de interface.

Por ejemplo:

```
interface Mamifero {
  String especie, color;
  public void mover();
}

class Gato implements Mamifero {
  int numero_patas;
  public void mover() {
    System.out.println("El gato es el que se mueve");
  }
}
```

En este ejemplo Gato no utiliza extends para hacer la herencia, sino implements.

Otro ejemplo

Runnable es un ejemplo de interface en el cual se declara, pero no se implemementa, una función miembro run.

```
public interface Runnable {
    public abstract void run();
}
```

Las clases que implementen (implements) el interface Runnable han de definir obligatoriamente la función run.

```
class Animacion implements Runnable{
       //..
       public void run(){
              //define la función run
       }
}
```

Clases que no están relacionadas pueden implementar el interface Runnable, por ejemplo, una clase que describa una animación, y también puede implementar el interface Runnable una clase que realice un cálculo intensivo.

Diferencias entre un interface y una clase abstracta

Un interface es simplemente una lista de métodos no implementados, además puede incluir la declaración de constantes. Una clase abstracta puede incluir métodos implementados y no implementados o abstractos, miembros dato constantes y otros no constantes.

Ahora bien, la diferencia es mucho más profunda.

El papel de la interface es el de describir algunas de las características de una clase. Por ejemplo, el hecho de que una persona sea un futbolista no define su personalidad completa, pero hace que tenga ciertas características que las distinguen de otras.

Una clase solamente puede derivar extends de una clase base, pero puede implementar varias interfaces. Los nombres de las interfaces se colocan separados por una coma después de la palabra reservada implements.

El lenguaje Java no fuerza por tanto, una relación jerárquica, simplemente permite que clases no relacionadas puedan tener algunas características de su comportamiento similares.

Por ejemplo, si quisiéramos crear una clase MiApplet que moviese una figura por su área de trabajo, tendríamos que hacer que esta clase se derive de la clase base Applet (que describe la funcionalidad mínima de un applet que se ejecuta en un navegador) y quisiéramos implementar la función run de la clase Runnable. Pero el lenguaje Java no tiene herencia múltiple.

En el lenguaje Java la clase MiApplet deriva de la clase base Applet e implementa el interface Runnable

```
class MiApplet extends Applet implements Runnable{
//...
//define la función run de la interface
public void run(){
//...
}
//redefine paint de la clase base Applet
public void paint(Graphics g){
//...
}
//define otras funciones miembro
}
```

ENLACE DINAMICO

El enlace dinámico es una característica clave del polimorfismo, permite resolver los comportamientos de un objeto en tiempo de ejecución.

En el lenguaje C, los identificadores de la función están asociados siempre a direcciones físicas antes de la ejecución del programa, esto se conoce como enlace temprano o estático. Ahora bien, el lenguaje C++ y Java permiten decidir a que función llamar en tiempo de ejecución, esto se conoce como enlace tardío o dinámico.

El enlace dinámico resuelve que método llamar en tiempo de ejecución cuando una clase de una jerarquía de herencia ha implementado el método. La maquina virtual Java observa el tipo de objeto para el cual se lleva a cabo la llamada, no el tipo de referencia al objeto del enunciado que llama.

El enlace dinámico también resuelve el manejo de los argumentos que se pasan a un método.

Como veremos más adelante se utilizará esta técnica para enlazar el nombre de un método con el código que se ejecutará para un objeto determinado, donde el método que finalmente se invocará en general sólo se conocerá durante la ejecución y no durante la compilación. Esto se conoce como el enlace dinámico de métodos.

IMPLEMENTACION DE POLIMORFISMO

El polimorfismo se implementa por medio de las funciones abstractas, en las clases derivadas se declara y se define una función que tiene el mismo nombre, el mismo número de parámetros y del mismo tipo que en la clase base, pero que da lugar a un comportamiento distinto, específico de los objetos de la clase derivada.

No se pueden crear objetos de una clase abstracta pero si se pueden declarar referencias en las que guardamos el valor devuelto por new al crear objetos de las clases derivadas. Esta peculiaridad nos permite pasar un objeto de una clase derivada a una función que conoce el objeto solamente por su clase base. De este modo podemos ampliar la jerarquía de clases sin modificar el código de las funciones que manipulan los objetos de las clases de la jerarquía.

Para implementar polimorfismo se utilizan los siguientes conceptos:

- Jerarquia de clases
- Métodos y clases abstractas
- Enlace dinámico y referencias

Utilizados de la siguiente manera:

1. Crear una jerarquía de clases con las operaciones importantes definidas por métodos que son abstractos en la clase base.
2. Proporcionar implementaciones concretas de las clases abstractas en las clases derivadas. Cada clase derivada tiene su propia versión de las funciones.
3. Manipular instancias de estas clases a través de una referencia, esto se debe a la ligadura dinámica utilizada en java.

Por ejemplo:

Creamos una clase abstracta denominada Animal de la cual deriva las clases Gato y Perro. Ambas clases redefinen la función habla declarada abstracta en la clase base Animal.

```
Public abstract class Animal {
    Public abstract void habla();
}

class Perro extends Animal{
    public void habla(){
        System.out.println("¡Guau!");
    }
}

class Gato extends Animal{
    public void habla(){
        System.out.println("¡Miau!");
    }
}
```

El polimorfismo se podría implementar de las siguientes formas:

1. Podemos pasar la referencia a un objeto de la clase Gato a una función hazleHablar que conoce al objeto por su clase base Animal.

```
Public class PoliApp2 {
  Public static void main(String[] args) {
    Gato gato=new Gato();
    hazleHablar(gato); }

  static void hazleHablar(Animal sujeto){
    sujeto.habla();
  }
}
```

2. Podemos crear un array de la clase base Animal y guardar en sus elementos los valores devueltos por new al crear objetos de las clases derivadas.

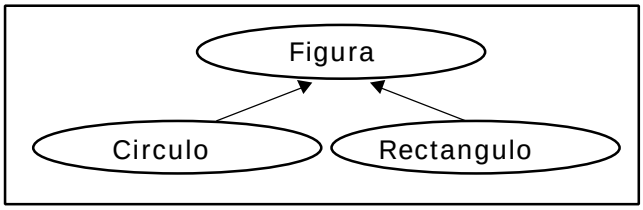
```
Public class PoliApp {
  Public static void main(String[] args) {
       Animal[] ani = new Animal[4];
       ani [0] =new Gato();
       ani [1]=new Perro();
       ani [2]=new Gato();
       ani [3]=new Perro();
       for (int i=0; i<4; i++)
              ani[i].habla();
       }
}
```

según sea el índice i se accederá al elemento del array que guarda una referencia a un objeto de la clase Gato o Perro.

Implementación de polimorfismo a través de un ejemplo

La jerarquía de clases que describen las figuras planas

Consideremos las figuras planas cerradas como el rectángulo, y el círculo. Tales figuras comparten características comunes como es la posición de la figura, de su centro, y el área de la figura, aunque el procedimiento para calcular dicha área sea completamente distinto. Podemos por tanto, diseñar una jerarquía de clases, tal que la clase base denominada Figura, tenga las características comunes y cada clase derivada las específicas. La relación jerárquica se muestra en la figura



La clase Figura es la que contiene las características comunes a dichas figuras concretas por tanto, no tiene forma ni tiene área. Esto lo expresamos declarando Figura como una clase abstracta, declarando la función miembro area abstract.

La clase Figura

La definición de la clase abstracta Figura, contiene la posición x e y de la figura particular, de su centro, y la función area, que se va a definir en las clases derivadas para calcular el área de cada figura en particular.

```
Public abstract class Figura {
    Protected int x;
    Protected int y;
    Public Figura(int x, int y) {
        this.x=x;
        this.y=y;
    }
    public abstract double area();
}
```

La clase Rectangulo

Las clases derivadas heredan los miembros dato x e y de la clase base, y definen la

función area, declarada abstract en la clase base Figura, ya que cada figura particular tiene una fórmula distinta para calcular su área. Por ejemplo, la clase derivada Rectangulo, tiene como datos, aparte de su posición (x, y) en el plano, sus dimensiones, es decir, su anchura ancho y altura alto.

```
class Rectangulo extends Figura{
    protected double ancho, alto;
    public Rectangulo(int x, int y, double ancho, double alto){
        super(x,y);
        this.ancho=ancho;
        this.alto=alto;
    }
    public double area(){
        return ancho*alto;
    }
}
```

La primera sentencia en el constructor de la clase derivada es una llamada al constructor de la clase base, para ello se emplea la palabra reservada super. El constructor de la clase derivada llama al constructor de la clase base y le pasa las coordenadas del punto x e y. Después inicializa sus miembros dato ancho y alto.

En la definición de la función area, se calcula el área del rectángulo como producto de la anchura por la altura, y se devuelve el resultado

La clase Circulo

```
class Circulo extends Figura{
    protected double radio;
    public Circulo(int x, int y, double radio){
        super(x,y);
        this.radio=radio;
    }
    public double area(){
        return Math.PI*radio*radio;
    }
}
```

Como vemos, la primera sentencia en el constructor de la clase derivada es una llamada al constructor de la clase base empleando la palabara reservada super. Posteriormente, se inicializa el miembro dato radio, de la clase derivada Circulo.

En la definición de la función area, se calcula el área del círculo mediante la conocida fórmula p*r2, o bien p*r*r. La constante Math.PI es una aproximación decimal del número irracional p.

Uso de la jerarquía de clases

Creamos un objeto c de la clase Circulo situado en el punto (0, 0) y de 5.5 unidades de radio. Calculamos y mostramos el valor de su área.

```
Circulo c=new Circulo(0, 0, 5.5);
System.out.println("Area del círculo "+c.area());
```

Creamos un objeto r de la clase Rectangulo situado en el punto (0, 0) y de dimensiones 5.5 de anchura y 2 unidades de largo. Calculamos y mostramos el valor de su área.

```
Rectangulo r=new Rectangulo(0, 0, 5.5, 2.0);
System.out.println("Area del rectángulo "+r.area());
```

Veamos ahora, una forma alternativa, guardamos el valor devuelto por new al crear objetos de las clases derivadas en una variable f del tipo Figura (clase base).

```
Figura f=new Circulo(0, 0, 5.5);
```

```
System.out.println("Area del círculo "+f.area());
f=new Rectangulo(0, 0, 5.5, 2.0);
System.out.println("Area del rectángulo "+f.area());
```

Enlace dinámico

A continuación veremos cómo se aplica el enlace dinamíco en nuestro ejemplo.

Podemos crear un array de la clase base Figura y guardar en sus elementos los valores devueltos por new al crear objetos de las clases derivadas.

```
Figura[] fig=new Figura[4];
fig[0]=new Rectangulo(0,0, 5.0, 7.0);
fig[1]=new Circulo(0,0, 5.0);
fig[2]=new Circulo(0, 0, 7.0);
fig[3]=new Rectangulo(0,0, 4.0, 6.0);
```

La sentencia

```
fig[i].area();
```

¿a qué función area llamará?. La respuesta será, según sea el índice i. Si i es cero, el primer elemento del array guarda una referencia a un objeto de la clase Rectangulo, luego llamará a la función miembro area de Rectangulo. Si i es uno, el segundo elemento del array guarda una referencia un objeto de la clase Circulo, luego llamará también a la función area de Circulo, y así sucesivamente. Pero podemos introducir el valor del índice i, a través del teclado, o seleccionando un control en un applet, en el momento en el que se ejecuta el programa. Luego, la decisión sobre qué función area se va a llamar se retrasa hasta el tiempo de ejecución.

El polimorfismo en acción

Supongamos que deseamos saber la figura que tiene mayor área independientemente de su forma. Primero, programamos una función que halle el mayor de varios números reales positivos.

```
double valorMayor(double[] x)
{
       double mayor=0.0;
       for (int i=0; i<x.length; i++)
              if(x[i]>mayor){
                     mayor=x[i];
              }
       return mayor;
}
```

Ahora, la llamada a la función valorMayor

```
double numeros[]={3.2, 3.0, 5.4, 1.2};
System.out.println("El valor mayor es "+valorMayor(numeros));
```

La función figuraMayor que compara el área de figuras planas es semejante a la función valorMayor anteriormente definida, se le pasa el array de objetos de la clase base Figura. La función devuelve una referencia al objeto cuya área es la mayor.

```
static Figura figuraMayor(Figura[] figuras){
        Figura mFigura=null;
        double areaMayor=0.0;
        for(int i=0; i<figuras.length; i++){
            if(figuras[i].area()>areaMayor){
                areaMayor=figuras[i].area();
                mFigura=figuras[i];
            }
        }
        return mFigura;
    }
```

La clave de la definición de la función está en las líneas

```
if(figuras[i].area()>areaMayor)
{
        areaMayor=figuras[i].area();
        mFigura=figuras[i];
}
```

En la primera línea, se llama a la versión correcta de la función area dependiendo de la referencia al tipo de objeto que guarda el elemento figuras[i] del array. En areaMayor se guarda el valor mayor de las áreas calculadas, y en mFigura, la figura cuya área es la mayor.

La principal ventaja de la definición de esta función estriba en que la función figuraMayor está definida en términos de variable figuras de la clase base Figura, por tanto, trabaja no solamente para una colección de círculos y rectángulos, sino también para cualquier otra figura derivada de la clase base Figura. Así si se deriva Triangulo de Figura, y se añade a la jerarquía de clases, la función figuraMayor podrá manejar objetos de dicha clase, sin modificar para nada el código de la misma.

Veamos ahora la llamada a la función figuraMayor

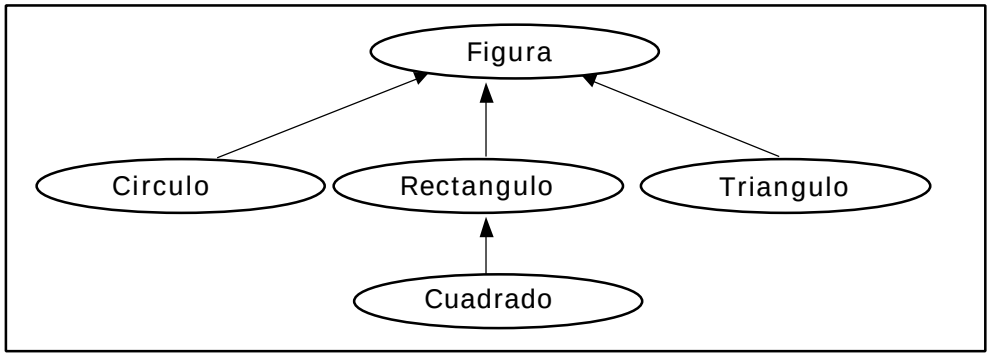
```
Figura[] fig=new Figura[4];
fig[0]=new Rectangulo(0,0, 5.0, 7.0);
fig[1]=new Circulo(0,0, 5.0);
fig[2]=new Circulo(0, 0, 7.0);
fig[3]=new Rectangulo(0,0, 4.0, 6.0);
Figura fMayor=figuraMayor(fig);
System.out.println("El área mayor es "+fMayor.area());
```

Pasamos el array fig a la función figuraMayor, el valor que retorna lo guardamos en fMayor. Para conocer el valor del área, desde fMayor se llamará a la función miembro area. Se llamará a la versión correcta dependiendo de la referencia al tipo de objeto que guarde por fMayor. Si fMayor guarda una referencia a un objeto de la clase Circulo, llamará a la función area definida en dicha clase. Si fMayor guarda una referencia a un objeto de la clase Rectangulo, llamará a la función area definida en dicha clase, y así sucesivamente.

La combinación de herencia y enlace dinámico se denomina polimorfismo. El polimorfismo es, por tanto, la técnica que permite pasar un objeto de una clase derivada a funciones que conocen el objeto solamente por su clase base.

Añadiendo nuevas clases a la jerarquía

Ampliamos el árbol jerárquico de las clases que describen las figuras planas regulares, para acomodar a dos clases que describen las figuras planas, triángulo y cuadrado. La relación jerárquica se muestra en la figura:



La clase Cuadrado

La clase Cuadrado es una clase especializada de Rectangulo, ya que un cuadrado tiene los lados iguales. El constructor solamente precisa de tres argumentos los que corresponden a la posición de la figura y a la longitud del lado

```
class Cuadrado extends Rectangulo{
    public Cuadrado(int x, int y, double dimension){
        super(x, y, dimension, dimension);
    }
}
```

El constructor de la clase derivada llama al constructor de la clase base y le pasa la posición x e y de la figura, el ancho y alto que tienen el mismo valor. No es necesario redefinir una nueva función area. La clase Cuadrado hereda la función area definida en la clase Rectangulo.

La clase Triangulo

La clase derivada Triangulo, tiene como datos, aparte de su posición (x, y) en el plano, la base y la altura del triángulo.

```
class Triangulo extends Figura{
    protected double base, altura;
    public Triangulo(int x, int y, double base, double altura){
        super(x, y);
        this.base=base;
        this.altura=altura;
    }
    public double area(){
        return base*altura/2;
    }
}
```

El constructor de la clase Triangulo llama al constructor de la clase Figura, le pasa las coordenadas x e y de su centro, y luego inicializa los miembros dato base y altura.

En la definición de la función area, se calcula el área del triángulo como producto de la base por la altura y dividido por dos.

El polimorfismo en acción

Veamos ahora la llamada a la función figuraMayor. Primero, creamos un array del tipo Figura, guardando en sus elementos las direcciones devueltas por new al crear cada uno de los objetos.

```
Figura[] fig=new Figura[4];
fig[0]=new Rectangulo(0,0, 5.0, 2.0);
fig[1]=new Circulo(0,0, 3.0);
fig[2]=new Cuadrado(0, 0, 5.0);
fig[3]=new Triangulo(0,0, 7.0, 12.0);

Figura fMayor=figuraMayor(fig);
System.out.println("El área mayor es "+fMayor.area());
```

Pasamos el array fig a la función figuraMayor, el valor que retorna lo guardamos en fMayor. Para conocer el valor del área, desde fMayor se llamará a la función miembro area. Se llamará a la versión correcta dependiendo de la referencia al tipo de objeto que guarda fMayor.

Si fMayor guarda una referencia a un objeto de la clase Circulo, llamará a la función area definida en dicha clase. Si fMayor guarda una referencia a un objeto de la clase Triangulo, llamará a la función area definida en dicha clase, y así sucesivamente.

PARTE PRACTICA
INTRODUCCION AL LENGUAJE JAVA

1. INTRODUCCION

Para aplicar los conceptos vistos en el teórico es preciso aprender los aspectos básicos del lenguaje Java: la primera aplicación, los comentarios, los tipos básicos de datos, los operadores, las sentencias condicionales e iterativas.

Java es un lenguaje orientado a objetos creado por James Gosling e introducido por Sun Mycrosystems en junio de 1995. Fue diseñado como un lenguaje de sintaxis muy similar a la del C++, pero con ciertas características diferentes que lo hacen más simple y portable a través de diferentes plataformas y sistemas operativos (tanto a nivel de código fuente como nivel binario), para lo cual se implementó como un lenguaje híbrido, a medio camino entre compilado e interpretado. A grandes rasgos, se compila el código fuente *java a un formato binario *.class *(bytecode),* que luego es interpretado por una máquina virtual java, la cual debe estar implementada para la plataforma particular usada.

En Java podemos construir dos tipos básicos de programas: utilizarlo como un lenguaje de propósito general para construir aplicaciones independientes o emplearlo para crear *applets* (tema que se verá en la unidad 2).

CARACTERISTICAS

Hay muchas razones por las que Java es tan popular y útil. Aquí se resumen algunas características importantes:

- **Orientación a objetos**: Java está totalmente orientado a objetos. No hay funciones sueltas en un programa de Java. Todos los métodos se encuentran dentro de clases. Los tipos de datos primitivos, como los enteros o dobles, tienen empaquetadores de clases, siendo estos objetos por sí mismos, lo que permite que el programa las manipule.
- **Simplicidad**: la sintaxis de Java es similar a ANSI C y C++ y, por tanto, fácil de aprender; aunque es mucho más simple y pequeño que C++. Elimina encabezados de archivos, preprocesador, aritmética de apuntadores, herencia múltiple, sobrecarga de operadores, struct, union y plantillas. Además, realiza automáticamente la recolección *de* basura, lo que hace innecesario el manejo explícito de memoria.
- **Robustez**: Por la forma en que está diseñado, Java no permite el manejo directo del hardware ni de la memoria (no permite modificar valores de punteros, por ejemplo), de modo que se puede decir que es virtualmente imposible "colgar" un programa Java. El intérprete siempre tiene el control. De hecho, el compilador es suficientemente inteligente como para no permitir una serie de acciones que podrían traer problemas, como usar variables sin inicializarlas, modificar valores de punteros directamente, acceder a métodos o variables en forma incorrecta, utilizar herencia múltiple, etc.
- **Compactibilidad**: Java está diseñado para ser pequeño. La versión más compacta puede utilizarse para controlar pequeñas aplicaciones. El intérprete de Java y el soporte básico de clases se mantienen pequeños al empaquetar por separado otras bibliotecas.
- **Portabilídad**: sus programas se compilan en el código de bytes de arquitectura neutra y se ejecutarán en cualquier plataforma con un intérprete de Java. Su compilador y otras herramientas están escritas en Java. Su intérprete está escrito en ANSI C. De cualquier modo, la especificación del lenguaje Java no *tiene* características *dependientes de* la implantación.
- **Amigable para el trabajo en red**: Java tiene elementos integrados para comunicación

en red, applets Web, aplicaciones cliente-servidor, además de acceso remoto a bases de datos, métodos y programas.

- **Soporte a GUI**: la caja de herramientas para la creación de ventanas abstractas - (Abstract Windowing Toolkit) de Java simplifica y facilita la escritura de programas GUI orientados a eventos con muchos componentes de ventana.
- **Carga y vinculación incremental dinámica**: las clases de Java se vinculan dinámicamente al momento de la carga. Por tanto, la adición de nuevos métodos y campos de datos a clases, no requieren de recompilación de clases del cliente.
- Internacionalización: los programas de Java están escritos en Unicode, un código de caracter de 16 bits que incluye alfabetos de los lenguajes más utilizados en el mundo. La manipulación de los caracteres de Unicode y el soporte para fecha/hora local, etcétera, hacen que Java sea bienvenido en todo el mundo.
- **Hilos**: Java proporciona múltiples flujos de control que se ejecutan de manera concurrente dentro de uno de sus programas. Los hilos permiten que su programa emprenda varias tareas de cómputo al mismo tiempo, una característica que da soporte a programas orientados a eventos, para trabajo en red y de animación.
- **Seguridad**: entre las medidas de seguridad de Java se incluyen restricciones en sus applets, implantación redefinible de sockets y objetos de administrador de seguridad definidos por el usuario. Hacen que las applets sean confiables y permiten que las aplicaciones implanten y se apeguen a reglas de seguridad personalizadas.

2. INSTALACION DE JAVA

Para configurar su ambiente Java, simplemente baje la versión mas reciente de JDK del sitio Web de JavaSoft:

http://www.javasoft.com/products/jdk/ e instálelo siguiendo las instrucciones.

PAQUETE DE CODIGO EJEMPLO JAVA

Un paquete de código ejemplo está disponible en el sitio web de la facultad en:

http://labsys.frc.utn.edu.ar, dirigirse a los sitios de las cátedra/PPR-2003/Unidad 1.

3. COMPILACION Y EJECUCION DE UN PROGRAMA

Vamos a describir cómo compilar y ejecutar el siguiente programa:

```
//Programa que muestra una frase en la pantalla.
import java.io.*;
public class EjemploSimple {
	public static void main(String args[]) {
		System.out.println ("Buenos Días");
	}
}
```

- Grabar el código en un archivo de texto. El programa fuente se guarda con el nombre de fichero EjemploSimple.java.
- Compilarlo en una ventana terminal (Unix) o de comandos (Windows) con "javac. *nombre_archivo.java".*

La compilación se lanza mediante el mandato javac seguido del nombre del fichero fuente:

```
C: \ProgramasJava\UnEjemplo>javac EjemploSimple.java
```

Esta instrucción provoca la compilación del fichero EjemploSimple.java, pero no crea un fichero ejecutable. Crea ficheros semicompilados, uno por clase definida en el fichero fuente (una sola en nuestro ejemplo); estos ficheros llevan el mismo nombre que la clase que describen, pero se caracterizan por la extensión .class.

- Ejecutarlo desde una ventana terminal (Unix) o de comandos (Windows) con "Java *nombre_archivo".*

  La ejecución del programa se lanza mediante el mandato:

  ```
  C: \Programasjava\UnEjemplo>java EjemploSimple
  ```

  Y éste es el resultado:

  ```
  Buenos Días
  ```

Sólo puede ejecutarse un programa en Java: el núcleo Java, al que hay que indicar cuál es la clase maestra de la aplicación. Así, en lugar de llamar a:

```
EjemploSimple
```

se llama a

```
java EjemploSimple
```

que lanza el núcleo Java y le indica que debe cargar la clase EjemploSimple, que encontrará en el fichero EjemploSimple.class, y ejecuta el método main de esta clase.

Este método debe declararse public static para poder ejecutarse.

EjemploSimple.class es una librería dinámica; no contiene código ejecutable como las librerías dinámicas clásicas, sino código Java, interpretado por el núcleo.

EL PROCESO EN MAS DETALLE

Los sistemas Java generalmente constan de varias partes: un entorno, el lenguaje, la interfaz de programación de aplicaciones (API, Applications Programming Interface) de Java y diversas bibliotecas de clases. A continuación analizaremos un entorno de creación de programas en Java representativo, el cual se ilustra a continuación.

Los programas Java normalmente pasan por cinco fases antes de ejecutarse. Éstas son: editar, compilar, cargar, verificar y ejecutar. En el laboratorio de sistema se utiliza el entorno JDK que sigue estrictamente estas especificaciones.

La fase 1 consiste en editar un archivo. Esto se hace con un programa editor. El programador teclea un programa en Java empleando el editor y hace correcciones si es necesario. A continuación el programa se almacena en un disposifivo de almacenamiento secundario, como un disco. Los nombres de archivo de los programas en Java terminan con la extensión .java. Dos editores que se utilizan mucho en los sistemas UNIX son vi y emace (en el laboratorio se encuentran disponibles, además de estos, otros editores). Los entornos de desarrollo de Java cuentan con editores incorporados que forman parte integral del entorno de programación. Suponemos que el editor sabe cómo editar un programa.

En la fase 2, el programador emite el comando javac para compilar el programa. El compilador de Java traduce el programa Java a códigos de bytes, que es el lenguaje que entiende el intérprete de Java. Si desea compilar un programa llamado HolaMundo.java, teclee:

javac HolaMundo.java

Si el programa se compila correctamcntc, sc producirá un archivo llamado HolaMundo.class. Éste es el archivo que contiene los códigos de bytes que serán interpretados durante la fase de ejecución.

La fase 3 se llama carga. Antes de que un programa pueda ejecutarse, es necesario colocarlo en la memoria. Esto lo hace un cargador de clases que toma el archivo (o archivos) .class que contiene los códigos de bytes y lo(s) transfiere a la memoria. El archivo .class puede cargarse de un disco de su sistema o a través de una red. El cargador de clases comienza a cargar archivos .class en dos situaciones. Por ejemplo, el comando:

java HolaMundo

Invoca el intérprete java para el programa HolaMundo y hace que el cargador de clases, cargue la información empleada en el programa HolaMundo. Llamamos al programa HolaMundo una aplicación. Las aplicaciones son programas que son ejecutados por el intérprete de Java. El cargador de clases también se ejecuta cuando una applet de Java se carga en un navegador de la World Wide Web, como Navigator de Netscape o Explorer de Microsoft. Las applets son programas de Java que normalmente se cargan de la Internet mediante un navegador. Las applets se cargan cuando el usuario curiosea un documento en HTML (Hypertext Markup Language, lenguaje de marcado de hipertexto) que contiene a la applet. Las applets también pueden ejecutarse desde la línea de comandos utilizando el comando appletviewer incluido en el Java Developer's Kit (JDK): un conjunto de herramientas que incluye al compilador (javac), al intérprete (java), a appletviewer y a otras herramientas empleadas por los programadores en Java. Al igual que Netscape Navigator y Explorer, appletviewer requiere un documento HTML. para invocar una applet. Por ejemplo, si el arehivo HolaMundo.html hace referencia a la applet HolaMundo, el comando appletviewer se usa así:

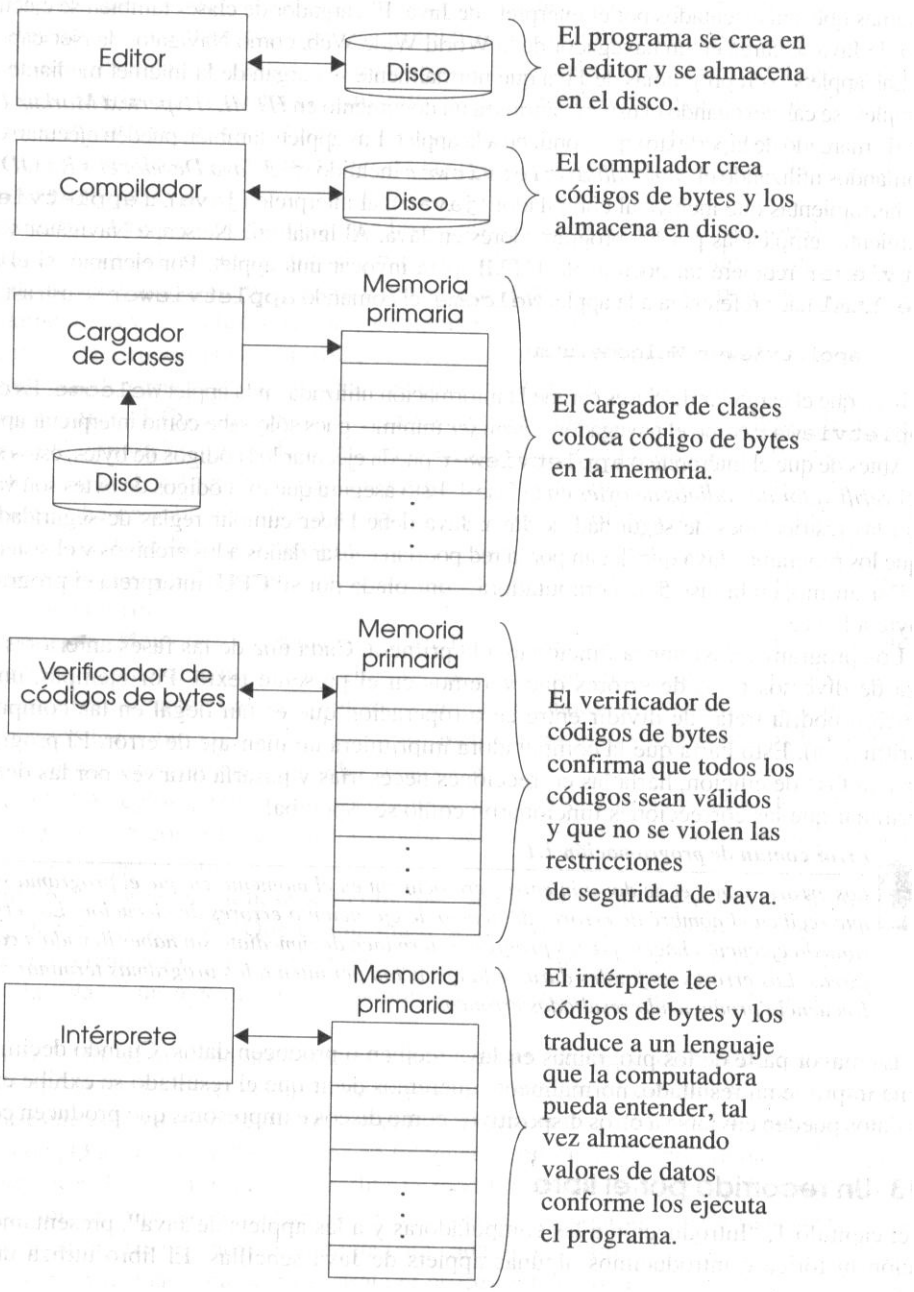
appletviewer HolaMundo.html

Esto hace que el cargador de clases cargue la infomación utilizada en la applet HolaMundo. Es común referirse a appletviewer como el navegador o browser mínimo, pues sólo sabe cómo interpretar applets.

Antes de que el intérprete o appletviewer pueda ejecutar los códigos de bytes, éstos son verificados por el verificador de códigos de bytes en la fase 4. Esto asegura que los códigos de bytes son válidos y que no violan las restricciones de seguridad de Java. Java debe hacer cumplir reglas de seguridad muy estrictas porque los programas Java que llegan por la red podrían causar daños a los archivos y el sistema del usuario.

Por último, en la fase 5, la computadora, controlada por su CPU, interpreta del programa, un código de byte a la vez.

Los programas casi nunca funcionan a la primera. Cada una de las fases anteriores puede fallar a causa de diversos tipos de errores. El programador regresaría a la fase de edición, haría las correcciones necesarias y pasaría otra vez por las demás fases para determinar que las correcciones funcionaron como se esperaba.



4. GRAMATICA

En general la gramática de java se parece a la del C/C++, vamos a ver los puntos más importantes para poder escribir un programa en java.

COMENTARIOS

En Java hay tres tipos de comentarios:

```
// comentarios para una sola línea

/* comentarios de una o
     más líneas
*/

/** comentario de documentación, de una o más líneas
*/
```

Los dos primeros tipos de comentarios son los que todo programador conoce y se

utilizan del mismo modo. Los comentarios de documentación, colocados inmediatamente antes de una declaración (de variable o función), indican que ese comentario ha de ser colocado en la documentación que se genera automáticamente cuando se utiliza la herramienta de Java, *javadoc*. Dichos comentarios sirven como descripción del elemento declarado permitiendo generar una documentación de nuestras clases escrita al mismo tiempo que se genera el código.

En este tipo de comentario para documentación, se permite la introducción de algunos tokens o palabras clave, que harán que la información que les sigue aparezca de forma diferente al resto en la documentación.

IDENTIFICADORES

Los identificadores nombran variables, funciones, clases y objetos; cualquier cosa que el programador necesite identificar o usar.

En Java, un identificador comienza con una letra, un subrayado (_) o un símbolo de dólar ($). Los siguientes caracteres pueden ser letras o dígitos. Se distinguen las mayúsculas de las minúsculas y no hay longitud máxima.

Serían identificadores válidos:

```
identificador
nombre_usuario
Nombre_Usuario
_variable_del_sistema
$transaccion
```

y su uso sería, por ejemplo:

```
int contador_principal;
char _lista_de_ficheros;
float $cantidad_en_Ptas;
```

Palabras clave

Las siguientes son las palabras clave que están definidas en Java y que no se pueden utilizar como indentificadores:

abstract	continue	for	new	switch
boolean	default	goto	null	synchronized
break	do	if	package	this
byte	double	implements	private	threadsafe
byvalue	else	import	protected	throw
case	extends	instanceof	public	transient
catch	false	int	return	true
char	final	interface	short	try
class	finally	long	static	void
const	float	native	super	while

Palabras reservadas

Además, el lenguaje se reserva unas cuantas palabras más, pero que hasta ahora no tienen un cometido específico. Son:

cast	future	generic	inner
operator	outer	rest	var

TIPOS DE DATOS

En Java existen 8 tipos primitivos de datos, que no son objetos, aunque el lenguaje cuenta con uno equivalente para cada uno de ellos. Los tipos básicos se presentan en la

siguiente tabla:

Tipo	**Tamaño / Formato**	**Descripción**
byte	8-bit complemento a 2	Entero de un byte
short	16-bit complemento a 2	Entero corto
int	32-bit complemento a 2	Entero
long	64-bit complemento a 2	Entero largo
float	32-bit IEEE 754	Punto flotante, precisión simple
double	64-bit IEEE 754	Punto flotante, precisión doble
char	16-bit caracter Unicode	Un carácter
boolean	true, false	Valor booleano (verdadero o falso)

El tamaño de estos tipos está fijado, siendo independiente del microprocesador y del sistema operativo sobre el que esté implementado. Esta característica es esencial para el requisito de la portabilídad entre distintas plataformas.

La razón de que se codifique los char con 2 bytes es para permitir implementar el juego de caracteres Unicode, mucho más universal que ASCII, y sus numerosas extensiones.

Java provee para cada tipo primitivo una clase correspondiente: Boolean, Character, Integer, Long, Float y Double.

¿Por qué existen estos tipos primitivos y no sólo sus objetos equivalentes? La razón es sencilla, por eficiencia. Estos tipos básicos son almacenados en una parte de la memoria conocida como el *Stack,* que es manejada directamente por el procesador a través de un registro apuntador *(stack pointer).* Esta zona de memoria es de rápido acceso, pero tiene la desventaja de que el compilador de java debe conocer, cuando está creando el programa, el tamaño y el tiempo de vida de todos los datos allí almacenados para poder generar código que mueva el *stack pointer,* lo cual limita la flexibilidad de los programas. En cambio, los objetos son creados en otra zona de memoria conocida como *Heap.* Esta zona es de propósito general y, a diferencia del *Stack,* el compilador no necesita conocer ni el tamaño, ni el tiempo de vida de los datos allí alojados. Este enfoque es mucho más flexible pero, en contraposición, el tiempo de acceso a esta zona es más elevado que el necesario para acceder al *stack.*

Aunque en la literatura se comenta que Java eliminó los punteros, esta afirmación es inexacta. Lo que no se permite es la aritmética de punteros. Cuando estamos manejando un objeto en Java, realmente estamos utilizando un *handie* a dicho objeto. Podemos definir un *handie* como una variable que contiene la dirección de memoria donde se encuentra el objeto almacenado.

VARIABLES

En un programa informático, hay que manipular los datos. Estos son accesibles, en general, mediante la utilización de:

parámetros de métodos o de funciones;
variables locales;
variables globales;
atributos de objetos.

En Java las variables globales no existen, el resto, parámetros, variables locales y atributos de objetos, se definen y manipulan de la misma forma que en C++.

Género de las variables

En informática hay dos maneras de almacenar las variables en un método:

- se almacena el valor de la variable;
- se almacena la dirección donde se encuentra la variable.

En Java:

- los tipos elementales se manipulan directamente: se dice que se manipulan por valor;
- los objetos se manipulan a través de su dirección: se dice que se manipulan por referencia.

Por ejemplo:

```
void Metodo() {
      int var1  = 2;
      Objeto var2 = new Objeto();
      var1  = var1  + 3;
      var2.agrega  (3);
}
```

En el ejemplo anterior, var1 representa físicamente el contenido de la variable mientras que var2 sólo representa la dirección que permite acceder a ella: se crea un objeto en algún lugar de la memoria y var2 representa físicamente el medio de acceder a él.

Cuando se añade 3 a var1, se modifica la variable var1, mientras que el método agrega (3) no modifica var2, sino el objeto designado por var2.

A priori preferirá var1, que le parecerá más simple. Es cierto, pero este mecanismo es mucho más restrictivo e impide por ejemplo compartir datos entre diferentes partes de la aplicación.

Además, el almacenamiento al estilo de var2 es indispensable cuando los objetos son algo más que tipos simples.

En Java, para todas las variables de tipo básico se accede al valor asignado a ellas directamente (no se conoce la dirección de memoria que ocupa). Para las demás (arrays, clases o interfaces), se accede a la variable a través de un puntero. El valor del puntero no es accesible ni se puede modificar como en C/C++; Java no lo necesita, además, esto atentaría contra la robustez del lenguaje.

De hecho, en Java no existen los tipos pointer, struct o union. Un objeto es más que una estructura, y las uniones no se hacen necesarias con un método de programación adecuado (además se evita la posibilidad de acceder a los datos incorrectamente).

Asignaciones a variables

Se asigna un valor a una variable mediante el signo =

Variable = Constante | Expresión;

Por ejemplo:

```
suma = suma + 1;
precio = 1.05 * precio;
```

Inicialización de las variables

En Java no se puede utilizar una variable sin haberla inicializado previamente. El compilador señala un error cuando se utiliza una variable sin haberla inicializado.

Veamos un ejemplo. La compilación de:

```
import java.io.*;
class DemoVariable {
      public static void main (String argv[]) {
            int noInicializado;
            System.out.println("Valor del entero: "+nolnicializado);
```

```
        }
    }
```

provoca el error siguiente:

```
c: \ProgramasJava\cramatica>javac init.java
init.java:6: Variable noInicializada may not have been initialized
System.out.println ("Valor del entero:"+ nolnicializado);

1 error
```

OPERADORES

Los operadores de Java son muy parecidos en estilo y funcionamiento a los de C. En la siguiente tabla aparecen los operadores que se utilizan en Java, por orden de precedencia:

operadores	
posfijos	[] . (parámetros) expr++ expr--
Operadores unarios	++expr --expr +expr –expr ~ !
Creación y "cast"	new (tipo)
Multiplicativos	* / %
Aditivos	+ -
Desplazamiento	<< >> >>>
Relacionales	< > <= >= instanceof
Igualdad	== !=
AND bit a bit	&
OR exclusivo bit a bit	^
OR inclusivo bit a bit	|
AND lógico	&&
OR lógico	||
Condicional	? :
Asignación	= += -= *= /= % = ^= &= |= <<= >>= >>>=

- Los operadores numéricos se comportan como esperamos. hay operadores unarios y binarios, según actúen sobre un solo argumento o sobre dos.

Operadores unarios

Incluyen, entre otros: +,-, ++, --, ~, !, (tipo)

Se colocan antes de la constante o expresión (o, en algunos casos, después). Por ejemplo:

```
-cnt;           //cambia de signo; por ejemplo si cnt es 12 el
                //resultado es -12 (cnt no cambia)
++cnt;          //equivalen a cnt+=1;
cnt++;
--cnt;          //equivalen a cnt-=1;
cnt--;
```

Operadores binarios

Incluyen, entre otros: `+, -, *, /, %`

Van entre dos constantes o expresiones o combinación de ambas. Por ejemplo:

```
cnt + 2                    //devuelve la suma de ambos.
promedio + (valor/2)
horas / hombres;           //división.
acumulado % 3;             //resto de la división entera entre ambos.
```

Nota: + sirve también para concatenar cadenas de caracteres. Cuando se mezclan Strings y valores numéricos, éstos se convierten automáticamente en cadenas:

```
"La frase tiene " + cant + " letras"
```

se convierte en:

```
"La frase tiene 17 letras"       //suponiendo que cant = 17
```

- Los operadores relacionales devuelven un valor booleano.
- El operador = siempre hace copias de objetos, marcando los antiguos para borrarlos, y ya se encargará el garbage collector de devolver al sistema la memoria ocupada por el objeto eliminado.

Veamos algunos ejemplos:

- **[ ]** define arreglos: `int lista [ ];`
- **(params)** es la lista de parámetros cuando se llama a un método: `convertir (valor, base);`
- **new** permite crear una instancia de un objeto: `new contador();`
- **(type)** cambia el tipo de una expresión a otro: `(float) (total % 10);`
- **>>** desplaza bit a bit un valor binario: `base >> 3;`
- **<=** devuelve "true" si un valor es menor o igual que otro: `total <= maximo;`
- **instanceof** devuelve "true" si el objeto es una instancia de la clase: `papa instanceof Comida;`
- || devuelve "true" si cualquiera de las expresiones es verdad: `(a<5) || (a>20).`

SEPARADORES

Sólo hay un par de secuencias con otros caracteres que pueden aparecer en el código Java; son los separadores simples, que van a definir la forma y función del código. Los separadores admitidos en Java son:

() - paréntesis. Para contener listas de parámetros en la definición y llamada a métodos. También se utiliza para definir precedencia en expresiones, contener expresiones para control de flujo y rodear las conversiones de tipo.

{} - llaves. Para contener los valores de matrices inicializadas automáticamente. También se utiliza para definir un bloque de código, para clases, métodos y ámbitos locales.

[ ] - corchetes. Para declarar tipos matriz. También se utiliza cuando se referencian valores de matriz.

; - punto y coma. Separa sentencias.

, - coma. Separa identificadores consecutivos en una declaración de variables. También se utiliza para encadenar sentencias dentro de una sentencia for.

. - punto. Para separar nombres de paquete de subpaquetes y clases. También se

utiliza para separar una variable o método de una variable de referencia.

ESTRUCTURAS DE CONTROL

Las estructuras de control en Java son básicamente las mismas que en C, con excepción del goto, que no existe.

if... [else]

La estructura más común de todas permite ejecutar una instrucción (o secuencia de instrucciones) si se da una condición dada (o, mediante la cláusula *else,* ejecutar otra secuencia en caso contrario).

```
if (expresión_booleana) instrucción_si_verdad
[ else instrucción_si_falso; ]
```

o bien:

```
if (expresión_booleana) {
        instrucciones_si_verdad;
}
else {
        instrucciones_si_falso;
}
```

Por ejemplo:

```
public final String toString() {
        íf (y<0)
                return x+"-i"+(-y);
        else
                return x+"+i"+y;
}
```

switch case.. .break. . .default

Permite ejecutar una serie de operaciones para el caso de que una variable tenga un valor entero dado. La ejecución saltea todos los *case* hasta que encuentra uno con el valor de la variable, y ejecuta desde allí hasta el final del case o hasta que encuentre un break, en cuyo caso salta al final del case. El default permite poner una serie de instrucciones que se ejecutan en caso de que la igualdad no se dé para ninguno de los case.

```
switch (expresión_entera) {
        case (valor1) : instrucciones_1; [break;]
        case (valor2) : instrucciones_2; [break;]
        ...
        case (valorN) : instrucciones_N; [break;]
        default: instrucciones_por_defecto;
}
```

Por ejemplo:

```
switch (Fecha.mes) {
        case (2):
                if (bisiesto(Fecha.anio)) dias=29;
                else días=28;
                break;
        case (4):
        case (6):
```

```
        case (9):
        case (11):
                días = 30;
                break;
        default:
                dias = 31;
}
```

while

Permite ejecutar un grupo de instrucciones mientras se cumpla una condición dada:

```
while (expresión_booleana) {
 instrucciones...
}
```

Por ejemplo:

```
while (linea != null) {
        linea = archivo.LeerLinea();
        System.out.println(linea);
}
```

do...while

Similar al anterior, sólo que la condición se evalúa al final del ciclo y no al principio:

```
do {
        instrucciones...
} while (expresión_booleana);
```

Por ejemplo:

```
do {
        linea = archivo.LeerLinea();
        if (linea != null)
                System.out.println(linea);
} while (linea != null);
```

for

También para ejecutar en forma repetida una serie de instrucciones, aunque es un poco más complejo:

```
for (instrucciones_iniciales;  condición_booleana; instrucción_repetitiva_x)  {
        instrucciones...
}
```

Si bien las instrucciones pueden ser cualesquiera (el bucle se repite mientras la condición sea verdadera), lo usual es utilizarlo para contar la cantidad de veces que dichas instrucciones se repiten. Se podría indicar de la siguiente manera:

```
for ( contador = valor_inicial; contador < valor_final; contador++ ) {
        instrucciones...
}
```

Por ejemplo:

```
for (i=0; i<l0; i++) {
        System.out.println( i );
}
```

Control General del Flujo

Cuando se necesita interrumpir el control del flujo se puede utilizar:

- **break** [etiqueta] : Permite saltar al final (break) de una ejecución repetitiva.
- **continue** [etiqueta] : Permite saltar al principio de una ejecución repetitiva.
- **return** expr; : Sale de una función.

Por ejemplo:

```
import java.io.*;
class bucles {
        public static void main(String argv[]) {
        int i=0;
        for (i=1; i<5; i++) {
        System.out.println("antes "+ i);
                if (i==2) continue;
                if (i==3) break;
        System.out.println("después "+i);
        }
}
```

La salida es:

```
antes 1
después 1
antes 2
antes 3
```

Otras Instrucciones

Hay otras instrucciones que controlan el flujo del programa:

- **catch**, **throw**, **try** y **finally** (para ver con las excepciones)
- **syncronized** (para ver junto con los threads)

5. UN PROGRAMA JAVA

Con los conceptos hasta ahora vistos ya podemos escribir un programa en Java.

Como primer paso crearemos un archivo de texto ASCII, al que llamaremos Hello.java. Un fichero de código fuente a menudo es llamado una unidad de compilación. En cada unidad de compilación puede existir como mucho una clase con el especificador de acceso public, el cual la hace visible al exterior del package. El resto de las clases permanecerán ocultas al exterior del package y su único objetivo será el de ayudar a la clase pública a realizar su cometido. El fichero de código fuente ha de ser llamado con el nombre de la clase pública (respetando las mayúsculas), seguido de la extensión java. Para compilar el fichero Hello.java, tecleamos desde la línea de comandos: javac hellojava

Si no hay errores obtendremos un fichero de extensión.class por cada clase definida en la unidad de compilación. En este caso obtenemos: Print.class y Hello.class. Para ejecutar el intérprete, tecleamos sin más: java Hello y obtendremos la salida por pantalla del programa.

```
//primer programa en Java
//el archivo se ha de llamar Hello.java
import java.util.Date;

class Print {
        static void prt(String s) {
                System.out.println(s);
        }
}
```

```
Public class Hello {
        //el punto de entrada del programa
        public static void main(String args[]) {
                System.out.println(new Date());
                String s1 = new String("Hola mundo");
                //otra manera de inicializar cadenas
                String s2= "Hola, de nuevo";
                Print.prt(sl);
                Print.prt(s2);
        }
}
```

Vamos a estudiar detalladamente este ejemplo:

La línea ímport java.utíl.Date; avisa al compilador de que quiero utilizar la clase standard Date en mí programa. El compilador por defecto importa el package java.lang, el cual contiene clases útiles de uso general, como la clase String, usada en este ejemplo. Siguiendo con el ejemplo, vemos definida una clase Príntt, la cual sólo contiene una función estática llamada prt, la cual no retorna ningún valor y que acepta como argumentos una cadena, que será impresa por el dispositivo de salida estándar (la pantalla). Para realizar esta tarea utilizamos un objeto estático out perteneciente a la clase System (incluida en java.lang), sobre el que ejecutamos el método println().

Siguiendo con el ejemplo nos encontramos con la definición de la clase Hello, que declara y define un único método. Cuando utilizamos Java para crear una aplicación de propósito general, una de las clases de la unidad de compilación debe llamarse como el fichero y además dicha clase debe tener definida una función de la forma:

```
public static void main(String args[])
```

Esta función es el punto de entrada del programa y nos está dando a entender que esta clase puede ser usada como módulo inicial en un programa. El argumento de este método es un array de cadenas que nos sirve para poder pasar parámetros al programa mediante la línea de comandos de forma bastante similar al C. Esta función es estática dado que va a ser llamada inicialmente al comenzar el programa, no existiendo en ese punto todavía ningún objeto creado. Esta función es pública porque tiene que poder ser accedida desde fuera del fichero. En la primera línea del cuerpo del método main nos encontramos con la sentencia System.out.println(new Date()); donde encontramos un par de ideas importantes:

- Aparece el concepto de sobrecarga de funciones que, básicamente, significa el hecho de que dos métodos tengan el mismo nombre, pero se diferencien en su lista de argumentos. En este caso el método println() está sobrecargado para poder aceptar como parámetro un objeto de tipo Date, encargándose el método de imprimir la fecha actual por la pantalla.
- Por otro lado, como parámetro de la función utilizamos un objeto creado mediante new, pero que no tiene asignado un handle. Es decir, va a ser un objeto temporal que, un cierto tiempo después de haber sido creado, será eliminado por el recolector de basura.

En las dos siguientes sentencias podemos comprobar el uso de una clase estándar para manejar cadenas constantes, llamada String. En primer lugar creamos el objeto mediante new, pasando como parámetro al constructor del objeto la cadena de caracteres que queremos manejar. El concepto de constructor será tratado ampliamente en el próximo capítulo así que, por ahora, nos basta con saber que es un método que contiene todas las clases y que facilita la inicialización de sus variables miembro. La clase String es un poco especial que nos permite usar una sintaxis similar a la del C++ para crear un objeto cadena:

```
String s2 = "Hola, de nuevo";
```

Las dos últimas sentencias del programa se limitan a llamar al método de clase prt con los argumentos apropiados (observar que no hemos necesitado crear ningún objeto

Print).

6. ARREGLOS

A primera vista, y sintácticamente hablando, los arreglos en Java no difieren verdaderamente de los arreglos del lenguaje C++. Para declarar un arreglo, se empleará una de las formas siguientes:

```
int [] VectorEnteros;
int VectorEnteros [];     // forma equivalente a la anterior
int VectorDeEnteros [][];
```

y para inicializarla:

```
int NumDiasFebrero[] = {28, 28, 28, 29};
string MombresDeMes [] = new string [12];
```

El acceso al arreglo se realiza de la forma siguiente:

```
NombresDeMes [0] = 'Enero';
System.out.println(NombresDeMes[0]);
```

Los arreglos son objetos, y como todo objeto, deben asignarse dinámicamente y no pueden existir estáticamente en el código.

```
int IntentoDeVectorEstatico [2];
```

Si se intenta compilar código que contenga la declaración anterior, se obtendrá el mensaje siguiente:

```
C:\programasjava\Gramatica>javac Vector.java
Vector.java:7: Can't specify array dimension in a declaration.
int IntentoDeVectorEstatico [2];

1 error
```

Sólo hay una forma para reservar espacio para un arreglo, y es asignarlo explícitamente. Así, el código escrito será más legible.

Además, los arreglos en Java gestionan su longitud, accesible por el método length(), y lanzan una excepción si interviene un desbordamiento. Veremos más adelante qué es una excepción. Sin embargo, recordemos que un error de desbordamiento de arreglo en C o en C++ conduce, en el mejor de los casos, a una parada del programa, y en la mayor parte de casos a resultados falsos.

En Java, el mecanismo de excepción hace que el error sea detectado por el núcleo Java, que pueda ser tratado eventualmente por el programa, pero en cualquier caso la salida del programa se realiza de forma controlada.

```
import java.io.*,

class Desbordamiento {
      public static void main (String argv[]) {
            int Vector [] = new int[4];
            System.out.println ("L: “ + Vector.length);
            Vector  [3]  = 3;
            Vector  [4]  = 4;
      }
}
```

dará en la ejecución:

```
C:\ProgramasJava\Gramatica>java Desbordamiento
L:4
java.lang.ArraylndexOutOfBoundsException: 4
at Desbordamiento.main(desbordamiento.java: 8)
```

El núcleo ha detectado el desbordamiento.

En Java, los arreglos se indexan mediante int. No pueden indexarse en ningún caso mediante long. La indexación empieza por 0, lo que significa que

```
Vector [0]
```

da acceso al primer elemento del arreglo, y que el acceso a

```
Vector [Vector.length()]
```

lanzará siempre una excepción, porque es un intento de acceder a un elemento que estaría justo después del fin de la matriz.

Pero queda pendiente una cuestión: las matrices de char, que como todo el mundo sabe son cadenas de caracteres, ¿deben terminarse con \O, y la longitud accesible por length tiene en cuenta este carácter terminal?

Esta cuestión no tiene sentido, porque las cadenas de caracteres no son arreglos. Son String. Un arreglo de char representa únicamente un arreglo que contiene char.

El código siguiente:

```
char mat [] = “ii”;
```

provoca un error del compilador, tal como muestra el mensaje siguiente:

```
e: \ProgramaaJava\Gramatica>javac matchar.java
matchar.java:5: Incompatible type for declaration. Can't convert
java.lang.string to char[].
char mat [] = 'ii";

1 error
```

7. CADENAS DE CARACTERES

Las cadenas de caracteres o string no son un tipo primitivo de datos, pero se pueden manejar a través de la clase String o StringBuffer, que son dos clases destinadas al manejo de cadenas de caracteres. La diferencia entre ambas es simple. La primera representa las cadenas constantes, mientras que la segunda representa las cadenas modificables.

Sin embargo, no crea que la clase String sólo sirve para almacenar cadenas escritas literalmente en el programa:

```
String cadenaConstante = "este cadena es constante.';
```

En efecto, un objeto de la clase String puede construirse con cualquier valor. Son posibles varios constructores, y sólo tras la construcción el objeto se convierte en intocable. Debido a que su longitud es constante, el núcleo Java sólo reserva para las cadenas constantes el espacio estrictamente necesario para su representación en memoria.

Por el contrario, las cadenas no constantes gestionan su longitud y el tamaño de su representación en memoria.

El programa.

```
import java.io. *

class TestStringBuffer {
	public static void main (String argv []) {
		StringBuffer cadenaTest = new StringBuffer();
		cadenaTest.append ("_123456789");
		System.out.println (cadenaTest +” ”+ cadenaTest.length()
						+ “, “ + cadenaTest.capacity ());
		cadenaTest.setLength (5);
		System.out.println (cadenaTest +” ”+ cadenaTest.length()
						+ “, “ + cadenaTest.capacity ());
	}
}
```

dará el resultado siguiente:

```
c:\ProgramasJava\gramatica>java TestStringBuffer
_123456769 10, 16
_1234 5, 16
```

La cadena varía en su longitud (pasa de 10 caracteres a 5), pero su capacidad no varía: el núcleo Java le ha asignado 16 de modo predeterminado, porque el constructor utilizado no especificaba capacidad, y se fija a 16 cuando la longitud de la cadena disminuye.

Esta noción de capacidad sólo es importante si desea gestionar con precisión los recursos de memoria de su sistema.

Recuerde pues las ventajas de los string en relación a los stringBuffer:

- son más económicos en recursos;
- son constantes y le garantizan que otro programa no pueda modificarlos.

El método System.out.println () sólo acepta cadenas constantes como parámetro, por lo que el compilador realiza una conversión implícita cuando le pasa como parámetro una cadena no constante.

Finalmente, observe que un strinqBuffer no puede transformarse en string, contrariamente a lo que el código siguiente podría dar a entender:

```
StringBuffer cadenal = new StringBuffer();
cadenal.append ("Este código es realmente justo");
System.out.println (cadenal.toString ());
```

La cadena no constante no se transforma en cadena constante, sino que da lugar a una cadena constante. Esta es un nuevo objeto, que no reemplaza al antiguo, sino que tiene su propia existencia separada de la cadena stringBuffer que la ha generado.

Observe que si el recogedor de basura de memoria no existiera, no se llegaría a gestionar estos objetos que son creados por efectos de desbordamiento. Pero con Java, el núcleo que gestiona la liberación de los objetos no utilizados, no hay que preocuparse de este problema.

Operaciones principales

Las operaciones que se pueden efectuar con un String son las siguientes:

- crear la cadena, gracias a media docena de constructores diferentes, que permiten construir una cadena vacía, una cadena a partir de otra, una cadena a partir de una serie de octetos, o incluso a partir de una cadena no constante;
- extraer un carácter particular;
- buscar un carácter o una subcadena en cl interior de la cadena;
- comparar dos cadenas;
- pasar los caracteres de minúsculas a mayúsculas y al revés;
- extraer los blancos;
- reemplazar caracteres;
- añadir otra cadena a la serie de la cadena actual.

Las operaciones de modificación no actúan sobre la propia cadena, sino que crean una nueva cadena.

Por ejemplo:

```
String una = "~MAYUSCULAS";
System.out.println (una.toLowerCase());
```

La línea anterior crea una segunda cadena a partir de la primera, que no sufre

cambios. Esta segunda cadena se utiliza para la llamada al método println( ), y después desaparece.

El número de operaciones diferentes que se pueden efectuar sobre cadenas constantes permite reducir el empleo de StringBuffer al mínimo.

Observe sin embargo lo que permite esta última clase. Se puede:

- añadir caracteres en el interior o al final de la cadena (no se produce la creación de una nueva cadena, sino que se modifica la cadena actual);
- añadir directamente a una cadena objetos de tipo int, boolean, etc., sin tener que convertirlos previamente en cadenas de caracteres.

En efecto, el ejemplo siguiente:

```
StringBuffer cadenaTest = new StringBuffer();
cadenaTest.append (true);
cadenaTest.append (“ ”);
cadenaTest.append (‘C’);
cadenaTest.append (2);
System.out.println (cadenaTest);
```

producirá el resultado siguiente:

```
true C2
```

Concatenación

Veamos una novedad respecto a C++, la concatenación de cadenas de caracteres:

```
short numTuristas = 0;
String tema = new string ();
//... cálculo del valor de las variables anteriores
System.out.printín (“Hay “+numTuristas+” personas presentes”+temal);
```

visualizará como resultado:

```
Hay 23 personas presentes en el seminario “Java y los operadores”.
```

8. REFERENCIAS EN JAVA

Las referencias en Java no son punteros ni referencias como en C++. Las referencias en Java son identificadores de instancias de las clases Java. Una referencia dirige la atención a un objeto de un tipo específico. No tenemos por qué saber cómo lo hace ni necesitamos saber qué hace ni, por supuesto, su implementación.

A continuación vamos a utilizar un ejemplo para demostrar el uso y la utilización que podemos hacer de las referencias en Java:

Pensemos en una referencia como si se tratase de la llave electrónica de la habitación de un hotel. Primero crearemos la clase **Habitacion**, implementada en el fichero Habitacion.java, mediante instancias de la cual construiremos nuestro Hotel:

```
public class Habitacion {
    private int numHabitacion;
    private int numCamas;

    public Habitacion() {
        habitacion( 0 );
        }

    public Habitacion( int numeroHab ) {
        habitacion( numeroHab );
        }

    public Habitacion( int numeroHab,int camas ) {
        habitacion( numeroHab );
        camas( camas );
        }
```

```
    public int habitacion() {
        return( numHabitacion );
        }

    public void habitacion( int numeroHab ) {
        numHabitacion = numeroHab;
        }

    public int camas() {
        return( camas );
        }

    public void camas( int camas ) {
        numCamas = camas;
        }
    }
```

El código anterior sería el corazón de la aplicación. Vamos pues a construir nuestro Hotel creando Habitaciones y asignándole a cada una de ellas su llave electrónica; tal como muestra el código siguiente, Hotel1.java:

```
public class Hotel1 {
    public static void main( String args[] ) {
        Habitacion llaveHab1;                       // paso 1
        Habitacion llaveHab2;

        llaveHab1 = new Habitacion( 222 );          // pasos 2, 3, 4 y 5
        llaveHab2 = new Habitacion( 1144,3 );
    }
}
```

Para explicar el proceso, dividimos las acciones en los cinco pasos necesarios para poder entrar en nuestra habitación.

El primer paso es la creación de la llave, es decir, definir la variable referencia, por defecto *nula*.

El resto de los pasos se agrupan en una sola sentencia Java. En dicha sentencia se producen los siguientes pasos: el operador **new** busca espacio para una instancia de un objeto de una clase determinada e inicializa la memoria a los valores adecuados. Luego invoca al método constructor de la clase, proporcionándole los argumentos adecuados. El operador **new** devuelve una referencia a sí mismo, que es inmediatamente asignada a la variable referencia.

Podemos tener múltiples llaves para una misma habitación:

```
. . .
Habitacion llaveHab3,llaveHab4;
llaveHab3 = llaveHab1;
llaveHab4 = llavehab2;
```

De este modo conseguimos copias de las llaves. Las habitaciones en sí mismas no se han tocado en este proceso. Así que, ya tenemos dos llaves para la habitación 222 y otras dos para la habitación 1144.

Una llave puede ser programada para que funcione solamente con una habitación en cualquier momento, pero podemos cambiar su código electrónico para que funcione con alguna otra habitación; por ejemplo, para cambiar una habitación anteriormente utilizada por un empedernido fumador por otra limpia de olores y con vistas al mar. Cambiemos pues la llave duplicada de la habitación del fumador (la 222) por la habitación con olor a sal marina, 1144:

```
. . .
llaveHab3 = llaveHab2;
```

Ahora tenemos una llave para la habitación 222 y tres para la habitación 1144. Mantendremos una llave para cada habitación en la conserjería, para poder utilizarla como

llave maestra, en el caso de que alguien pierda su llave propia.

Alguien con la llave de una habitación puede hacer cambios en ella, y los compañeros que tengan llave de esa misma habitación, no tendrán conocimiento de esos cambios hasta que vuelvan a entrar en la habitación. Por ejemplo, vamos a quitar una de las camas de la habitación, entrando en ella con la llave maestra:

```
. . .
llaveHab2.camas( 2 );
```

Ahora cuando los inquilinos entren en la habitación podrán comprobar el cambio realizado:

```
. . .
llaveHab4.printData();
```

REFERENCIAS Y ARRAYS

Como en C y C++, Java dispone de arrays de tipos primitivos o de clases. Los arrays en C y C++ son básicamente un acompañante para los punteros. En Java, sin embargo, son ciudadanos de primera clase.

Vamos a expandir nuestro hotel creando todo un ala de habitaciones, Hotel2.java. Crearemos un juego de llaves maestras y luego construiremos las habitaciones:

```
public class Hotel2 {
    // Número de habitaciones por ala
    public static final int habPorAla = 12;

    public static void main( String args[] ) {
        Habitacion llavesMaestras[];                    // paso 1
        llavesMaestras = new Habitacion[ habPorAla ];   // pasos 2-5
//      ^^^^^^^^^^^^^^   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
//            A                 B, C, D y E
        int numPiso = 1;
        for( int i=0; i < habPorAla; i++ )              // pasos 6-9
            llavesMaestras[ i ] = new Habitacion( numPiso * 100 + i,
                ( 0 == (i%2)) ? 2 : 1 );
        for( int i=0; i < habPorAla; i++ )              // pasos 10-11
            llavesMaestras[i].printData();
        }
    }
```

Cada paso en el ejemplo es semejante al que ya vimos antes. El paso 1 especifica que el juego de llaves maestras es un grupo de llaves de habitaciones.

Los pasos 2 a 5 son, en este caso, la parte principal. En lugar de crear una habitación, el gerente ordena construir un grupo contiguo de habitaciones. El número de llaves se especifica entre corchetes y todas se crean en blanco.

Los pasos 6 a 9 son idénticos a los pasos 2 a 5 del ejemplo anterior, excepto en que en este caso todas las llaves pasan a formar parte del juego maestro. Los números de piso se dan en miles para que cuando se creen las habitaciones, todas tengan el mismo formato. También todas las habitaciones de número par tienen una sola cama, mientras que las habitaciones impares tendrán dos camas.

Los pasos 10 y 11 nos permiten obtener información de cada una de las habitaciones.

REFERENCIAS Y LISTAS

Retomemos el ejemplo de los arrays, y en vez de éstos vamos a usar una lista enlazada. El paquete de la lista simple se compone de dos clases. Cada elemento de la lista es un `NodoListaEnlazada`, NodoListaEnlazada.java:

```
public class NodoListaEnlazada {
    private NodoListaEnlazada siguiente;
```

```
        private Object datos;
        // . . .
    }
```

Cada NodoListaEnlazada contiene una referencia al nodo que le sigue. También contiene una referencia genérica a cualquier clase que se use para proporcionar acceso a los datos que el usuario proporcione.

La lista enlazada, ListaEnlazada.java, contiene un nodo raiz y un contador para el número de nodos en la lista:

```
    public class ListaEnlazada {
        public NodoListaEnlazada Raiz;
        private int numNodos;
        // . . .
        }
```

Revisemos pues el código de nuestro Hotel, ahora Hotel3.java, que será prácticamente el mismo que en el caso de los arrays:

```
    public class Hotel3 {
        // Número de habitaciones por ala
        public static final int habPorAla = 12;

        public static void main( String args[] ) {
            ListaEnlazada llaveMaestra;                     // paso 1
            llaveMaestra = new ListaEnlazada();             // pasos 2-5

            int numPiso = 1;
            for( int i=0; i < habPorAla; i++ )              // pasos 6-9
                llaveMaestra.insertAt(i,
                        new Habitacion(numPiso*100+i, (0==(i%2))?2:1 );
            for( int i=0; i < habPorAla; i++ )              // pasos 10-12
                ( (Habitacion)llaveMaestra.getAt(i) ).printData();
            }
        }
```

El paso 1 es la llave maestra de la lista. Está representada por una lista genérica; es decir, una lista de llaves que cumple la convención que nosotros hemos establecido. Podríamos acelerar el tiempo de compilación metiendo la lista genérica ListaEnlazada dentro de una ListaEnlazadaHabitacion.

Los pasos 2 a 5 son equivalentes a los del primer ejemplo. Construimos e inicializamos una nueva ListaEnlazada, que usaremos como juego de llaves maestras.

Los pasos 6 a 9 son funcionalmente idénticos a los del ejemplo anterior con arrays, pero con diferente sintaxis. En Java, los arrays y el operador [] son internos del lenguaje. Como Java no soporta la sobrecarga de operadores por parte del usuario, tenemos que usarlo siempre en su forma normal.

La ListaEnlazada proporciona el método *insertAt()* para insertar un elemento en la lista, tiene dos argumentos el primero indica el índice en la lista, donde el nuevo nodo ha de ser insertado y el segundo argumento es el objeto que será almacenado en la lista. Obsérvese que no es necesario colocar moldeo alguno para hacer algo a una clase descendiente que depende de uno de sus padres.

Los pasos 10 a 12 provocan la misma salida que los pasos 10 y 11 del ejemplo con arrays. El paso 10 toma la llave del juego que se indica en el método *getAt()*. En este momento, el sistema no sabe qué datos contiene la llave, porque el contenido de la habitación es genérico. Pero nosotros sí sabemos lo que hay en la lista, así que informamos al sistema haciendo un moldeado a la llave de la habitación (este *casting* generará un chequeo en tiempo de ejecución por el compilador, para asegurarse de que se trata de una Habitacion). El paso 12 usa la llave para imprimir la información

9. PAQUETES

El paquete (package)

Los paquetes son una forma de organizar grupos de clases. Un paquete contiene un conjunto de clases relacionadas bien por finalidad, por ámbito o por herencia.

Los paquetes resuelven el problema del conflicto entre los nombres de las clases. Al crecer el número de clases crece la probabilidad de designar con el mismo nombre a dos clases diferentes.

Las clases tienen ciertos privilegios de acceso a los miembros dato y a las funciones miembro de otras clases dentro de un mismo paquete.

En el Entorno Integrado de Desarrollo (IDE) JBuilder de Borland, un proyecto nuevo se crea en un subdirectorio que tiene el nombre del proyecto. A continuación, se crea la aplicación, un archivo .java que contiene el código de una clase cuyo nombre es el mismo que el del archivo. Se pueden agregar nuevas clases al proyecto, todas ellas contenidas en archivos .java situadas en el mismo subdirectorio. La primera sentencia que encontramos en el código fuente de las distintas clases que forman el proyecto es package o del nombre del paquete.

```
//archivo MiApp.java

package nombrePaquete;
public class MiApp{
       //miembros dato
       //funciones miembro
}
//archivo MiClase.java

package nombrePaquete;
public class MiClase{
       //miembros dato
       //funciones miembro
}
```

La palabra reservada import

Para importar clases de un paquete se usa el comando import. Se puede importar una clase individual

```
import java.awt.Font;
```

o bien, se puede importar las clases declaradas públicas de un paquete completo, utilizando un arterisco (*) para reemplazar los nombres de clase individuales.

```
import java.awt.*;
```

Para crear un objeto fuente de la clase Font podemos seguir dos alternativas

```
import java.awt.Font;
Font fuente=new Font("Monospaced", Font.BOLD, 36);
```

O bien, sin poner la sentencia import

```
java.awt.Font fuente=new java.awt.Font("Monospaced", Font.BOLD, 36);
```

Normalmente, usaremos la primera alternativa, ya que es la más económica en código, si tenemos que crear varias fuentes de texto.

Se pueden combinar ambas formas, por ejemplo, en la definición de la clase BarTexto

```
import java.awt.*;
public class BarTexto extends Panel implements java.io.Serializable{
//...
}
```

Panel es una clase que está en el paquete java.awt, y Serializable es un interface que está en el paquete java.io

Los paquetes estándar

Paquete	Descripción
java.applet	Contiene las clases necesarias para crear applets que se ejecutan en la ventana del navegador.
java.awt	Contiene clases para crear una aplicación GUI independiente de la plataforma.
java.io	Entrada/Salida. Clases que definen distintos flujos de datos.
java.lang	Contiene clases esenciales, se importa impícitamente sin necesidad de una sentencia import.
java.net	Se usa en combinación con las clases del paquete java.io para leer y escribir datos en la red.
java.util	Contiene otras clases útiles que ayudan al programador.

Resumen

- Las clases pueden organizarse en **paquetes**
- Un paquete se identifica con la cláusula **pachage algún paquete**, al principio de un archivo fuente.
- Solo una declaración **pachage** por archivo.
- Varios archivos pueden pertenecer al mismo **paquete**
- Un archivo sin declaración **pachage** pertenece al **paquete unnamed**
- Los nombres de los paquetes están relacionados con la organización de los archivos
- Los directorios padre de los paquetes deben incluirse en CLASSPATH
- Al compilar o ejecutar, las clases de biblioteca proporcionadas por Java (Java, Javac, JDK) se cargan automáticamente, otras deben especificarse en CLASS PATH.

  SET CLASSPATH= C:\JAVA; //Hasta el directorio padre de paquetes
- Para incorporarlas en mi programa, puedo

  import mis Paquetes.* // Todas las clases que alli existan

  import mis Paquetes.Madre // Solo la clase madre
- Para nombres de paquetes, recomendable usar nombres de dominio de Internet
- La compilación de un paquete produce tantos archivos .class como clases tenga el paquete.
- Los archivos .class se graban en el mismo directorio del fuente .Java, a menos que la opción –d indique otro destino

10. MANEJO BASICO DE ERROR Y EXCEPCION

Si un método encuentra un error durante la ejecución, puede retornar un valor predefinido como error. Para ello, debe hacer internamente una consistencia. Ejemplo, el método indexof (char c) de una cadena: Retorna -1 si el carácter no existe dentro de la cadena.

Trabajar de esta manera significa que debemos tener una buena cantidad de código distribuida en nuestros métodos, preocupándose de la calidad de los datos. Si bien esto ocurre con frecuencia, no es una forma racional ni eficiente de encarar este problema.

La manera adecuada y elegante de hacerlo es mediante lo que se llama "Tratamiento de Excepciones". Muchos lenguajes de programación (desde hace tiempo) lo soportan, C++ y Java también lo tienen.

Tratar excepciones es una manera sistemática de representar, transmitir, capturar y manejar errores concretos ocurridos durante la ejecución del programa. Entre ellos podemos citar:

- división por cero
- un argumento fuera de su posible dominio
- resultados fuera del rango permitido
- argumentos no esperados
- referencias ilegales
- desborde de índice de arreglo
- archivo no existente o inaccesible
- error de E/S.

EXCEPCIONES EN JAVA

Java implementa excepciones para facilitar la construcción de código robusto. Cuando ocurre un error en un programa, el código que encuentra el error lanza una excepción, que se puede capturar y recuperarse de ella. Java proporciona muchas excepciones predefinidas.

Las excepciones se representan en Java como objetos de clases que extienden, directa o indirectamente, la clase java.lang.Exception.

Java.lang.Object

Java.lang.Trowable

Java.lang.Exception

Son subclases de Exception, entre otras (más de 40 subclases)

AclNotFoundException, Excepción disparada cuando se hace una referencia a un ACL (Access Control List) inexistente

ActivationException, Excepción de tipo general usada por la interface de activación.

AlreadyBoundException, Excepción disparada cuando se hace un intento de unir un objeto en registro a un nombre que ya tiene vinculaciones activas.

AWTException, Señala que una excepción tipo Abstract Window Toolkit ha ocurrido

BadLocationException, Informa sobre intento de alcanzar localizaciones inexistentes ...

...

IOException, Se ocupa de las excepciones de entrada/salida. Es la superclase de estas excepciones y la extienden 16 sub clases.

→ **FileNotFoundException**, Una de ellas. Señala que un intento de abrir el "camino/archivo" ha fracasado.

Exception tiene un par de constructores.

Exception() Construye una excepción sin mensaje asociado.

Exception(String s) Construye una excepción con mensaje asociado.

Hereda métodos de Throwable :

fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace, printStackTrace, printStackTrace, toString

Hereda métodos de Object:

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Podemos además definir nuestras propias clases de tratamiento de errores específicos.(Es lo que haremos cuando tratemos nuestra clase ejemplo Matrix)

Las palabras clave para tratar excepciones son: **try, catch, throw, throws y finally**

CAPTURA DE EXCEPCIONES

La palabra clave **try** (tentativa, ensayo, probar) controla un *bloque de prueba* (instrucción compuesta) cuyas excepciones pueden capturarse y manejarse con un código asociado con la palabra clave catch. El formato general:

```
try
       { instrucciones } // Bloque de prueba

catch  (tipo1 e)          //  Cláusula de captura
       {  instrucciones }  //  Tratamiento excepciones tipo 1
catch(tipo2 e)
       {instrucciones }    //  Tratamiento excepciones tipo 2
       .  .  .  .  .
finally
       { instrucciones }   //  Ejecutado si ocurre excepción en bloque de prueba
```

Supongamos que *instrucciones* del bloque de prueba contenga una cadena de llamadas a métodos. Esto es normal, un método resuelve una fracción del problema, lo pasa o otro, el cual hace lo mismo ... tenemos una cadena de llamadas. Supongamos que en el tercero ocurre una excepción. Entonces se genera un objeto de ese tipo de excepción específica. El flujo de control retrocede hasta el inicio de la cadena, la palabra try. A continuación se verifica si el tipo de excepción generado esta declarado en alguna de la lista de cláusulas catch que siguen a try. Cada una de ellas tiene la forma de una definición de método con una lista de parámetros y un cuerpo de instrucciones. Sólo se capturan las excepciones cuyo tipo coincide con el del parámetro. El objeto de excepción se pasa al parámetro e de la primera cláusula y se ejecuta la instrucción. Puede ocurrir que el tipo de excepción no sea ninguno de la lista de captura. Entonces la excepción no es capturada aquí. Las no capturadas se pasan aún más arriba en la cadena de llamadas, o sea a

métodos que llamaron al método que contenía el try donde se disparó la excepción. En este retroceso se inspeccionan todas las listas de captura asociadas a cada cláusula try, tratando de encontrar una coincidencia de tipo. Si no llega a establecerse coincidencia, el retroceso llaga hasta la máquina virtual Java, el entorno donde corre nuestra aplicación.

El siguiente método trata de llevar a minúsculas los caracteres del flujo de entrada.

```
public static void doio(Inputstream in, Outputstrean out)
{
     int c;
     try {
             while ((c = in.read()) >= 0)
             {  c = Character.toLowercase((char) c);
                out.write(c);
         }
     catch (IOException e)
         {   Systen.err.println("doio: I/O problen");
             System.exit (1);
         }
}
```

Note que la ejecución continuará normalmente después del mensaje, ya que el programa no llama a System.exit.

La cláusula finally siempre se ejecuta si ocurre una excepción en el bloque de prueba. Utilice la cláusula finally para proporcionar acciones de limpieza necesarias en cualquier caso. Por ejemplo si la parte try abre flujos de archivo y usted quiere asegurarse de que estos se cierren adecuadamente, entonces use un código como:

```
try {
       FileInputstream infile = new FileInputStream(args[0]);
       File tmp_file = new File(tmp_name);
       // ...
    }
catch (FileNotFoundException e )
    {   System.err.println("Can't open input file   + args[0]);
        error = true;
    }
catch ( IOException e)
    {   System.err.println("Can't open temporary file" + tmp_name);
        error = true;
    }
finally
    {   if ( infile != null ) infile.close()>;
        if ( tmp_file != null ) tmp_file.delete();
        if ( error ) System.exit(1);
    }
```

TIPOS DE EXCEPCIONES

La clase Throwable (: Capaz de ser lanzada) del paquete java.lang es la superclase final de cualquier objeto de excepción. Tenemos dos familias de excepciones.

Excepciones a nivel máquina virtual Java*:* se interpreta al momento de la ejecución el código de bytes y se detectan: referencia a un puntero nulo, división por cero, desborde de indexación de arreglos. Estas excepciones son tratadas por las clases RuntimeException y Error.

Excepciones verificadas: son las restantes. Son empleadas por el código de usuario y son parte de la interfaz de llamada a método. Un método especifica excepciones

verificadas que pueden ser disparadas por la cláusula throws en el encabezado del método.

Un ejemplo, el mismo método doio, ahora usando throws.

```
public static void doio(InputStream in, OutputStream out)
            throws IOException
{
       mt c;
       while ( (c = in.read()) >= 0)
      {
              c = Character.toLowercase( (char) c);
              out.write(c);
      }
}
```

En lugar de capturar la excepción IOException como antes, ahora simplemente la especificamos como lanzable. (cláusula throws).

En definitiva, en los programas de Java, suponiendo que Ud. use tratamiento de excepciones (recomendable), la posible excepción verificada debe:

- capturarse en el método, (cláusula catch, primer ejemplo doio)
- especificarse mediante cláusula throws tipo1, tipo2, ... (Seg. ejemplo doio)

Finalmente, es posible generar excepciones propias, a voluntad. Una aplicación:

Supongamos que se desee retroceder abruptamente en la cadena de llamadas a método a un punto específico. Como vimos, en las excepciones generadas en try no capturadas por su lista de cláusulas catch el control retrocedía hasta encontrar alguna con el tipo de excepción adecuado(o terminaba en la máquina virtual). Este mecanismo puede ser aprovechado. Lanzamos una excepción de un tipo específico. Por ejemplo:

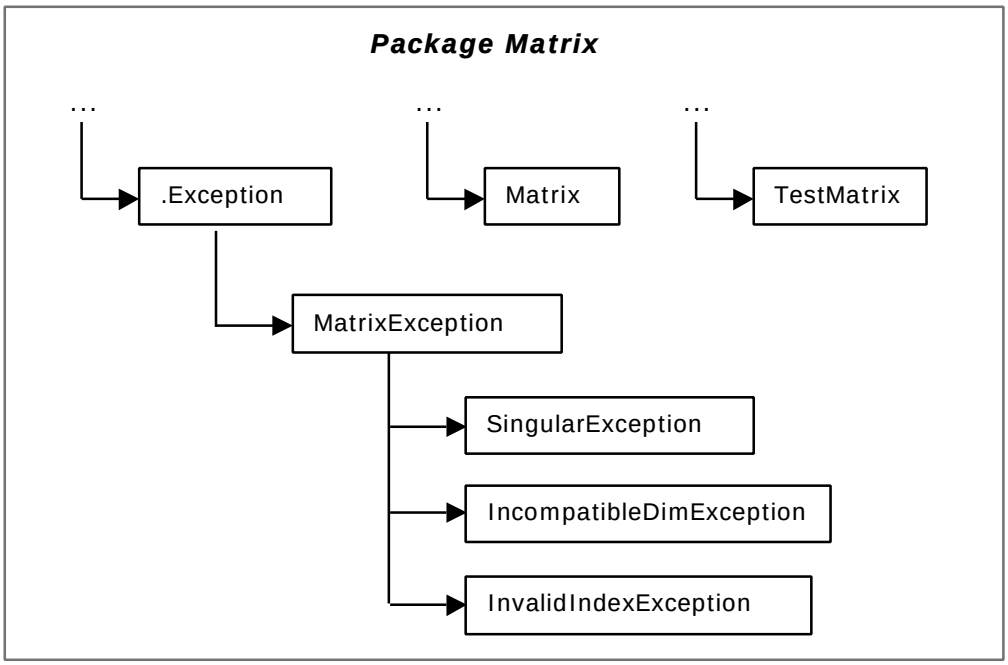
throw new IOException(); // o también

throw new IOException("Archivo no existe (???)");

El sistema la buscará, y si hemos programado bien la encontrará en una cláusula catch para IOException situada donde queremos. Habremos logrado el retorno deseado.

UN EJEMPLO QUE MANEJA EXCEPCIONES

Concluyamos con un ejemplo que trata eficientemente (y didácticamente) excepciones; las clases conforman el paquete Matrix y están organizadas:




```
///////    Matrix.java   Inicio ///////
package matrix;   //  A él pertenecen ...

public class Matrix  {

       private double[][] mat; // the matrix
       private int nr, nc;     // rows and cols

       public Matrix(int r, int c) {                 // Un constructor
         if ( r > 0 && c > 0 )
                    mat = new double[nr=r][nc=c];
              else {
                    mat = null; nr=nc=0;
              }
         }

       public Matrix(double[][] m) {                 // Otro
         mat = m;
         nr = m.length;
         nc = m[0].length;
       }

       protected Matrix() { }                        // El último ...

       // de este tipo pueden ocurrir excepciones
       public double element(int i, int j) throws InvalidIndexException
       {
         if ( i < 0 || j < 0 || i >= nr || j >= nc )
              throw new InvalidIndexException(
                    "Matrix element(" + i + "," + j + ")" );
          //  Y ocurrieron ...
          return mat[i][j]; // el que quería ...
        }

       // Una sobrecarga para incluir
       public void element(int i, int j, double e) throws InvalidIndexException
       {
         if ( i < 0 || j < 0 || i >= nr || j >= nc )
              throw new InvalidIndexException(
                      "Matrix element(" + i + "," + j + "," + e + ")" );
         mat[i][j] = e; // Hemos incluido ...
```

```
        }

        public int rows() { return nr; }
        public int cols() { return nc; }

        public Matrix times(Matrix b) throws MatrixException       // Producto
        {
          if ( nc != b.nr )
               throw new IncompatibleDimException(
                      "Matrix times: "+ nr+"X"+nc + " and "+ b.nr+"X"+b.nc);
               /* Observese que lanzamos (throw) una excepción distinta a las
               previstas como "lanzables" (throws). Pero es una sub clase. Todo
               bien. */
           Matrix p = new Matrix(nr,b.nc);
          for (int i=0 ; i < nr ; i++)
          {
               for (int j=0 ; j < b.nc; j++)
                      p.element(i,j,
               rowTimesCol(i, b.mat, j));
           }
           return(p); // Matrix producto
         }

         public String toString()                            // Para impresión
         {
           StringBuffer buf = new StringBuffer();
            for (int i = 0; i < nr; i++)
            {
               buf.append("( ");
               for (int j = 0; j < nc-1; j++)
               {
                      buf.append(mat[i][j]);
                      buf.append("  ");
               }
               buf.append(mat[i][nc-1]);
               buf.append(" )\n");
            }
           buf.setLength(buf.length()-1);
           return new String(buf);
        }

        private double rowTimesCol(int ri,double[][] b, int ci)
        {
          double sum=0.0d;
          double[] row = mat[ri];
           for (int k=0; k < b.length; k++)
               sum += row[k] * b[k][ci];
          return(sum);
        }
}
///////    Matrix.java   Fin ///////

///////    TestMatrix.java    Inicio ///////
package matrix;

class TestMatrix {
        public static void main(String[] args) throws ClassNotFoundException

         /*  Preveemos pueden producirse excepciones del tipo tratado por esta
clase heredera de Exception. Se producen estas excepciones cuando
usamos el nombre de la clase para invocar:
El método forName en la clase Class
findSystem en la clase ClassLoader
loadClass en la clase ClassLoader,
pero no tenemos la citada clase
Un ejemplo: la siguiente sentencia
(SortAlgorithm)Class.forName(algName).newInstance();
(class SortItem, demo de ordenamiento, applets)  */
```

```
        {
                double[][] a = { {1.0,-2.0,1.5}, {1.0,2.4,3.1}};
                //  dos filas, tres col.
                double[][] b = { {9.0,0.7},{-2.0,30.5},{-9.0,4.0}};
                //  tres filas, dos columnas
                double[][] c = { {9.0,0.7},{-2.0,30.5}};

                //  cuadrada dos por dos
                Matrix m1 = new Matrix(a);
                Matrix m2 = new Matrix(b);
                Matrix m3 = new Matrix(c);
                try {
                        Matrix prod = m1.times(m2);
                        /* Recordemos que el método times contiene "throws
                        MatrixException", entonces si esta excepción no se dio
                        imprimimos la matriz ...          */

                        System.out.println(prod);
                        prod = m1.times(m3);
                }
                catch(MatrixException e)
                {

                        /* En cambio, si algo anduvo mal invocamos nuestro gestor de
                        excepciones quien las trata, y nos vamos ...   */
                        handle(e);
                        System.exit(1);
                }
        }

        private static void handle(MatrixException e)
        throws ClassNotFoundException
        {
                if ( e instanceof SingularException )
                {
                        System.err.println("S:" + e.getMessage());
                }
                else if ( e instanceof IncompatibleDimException )
                {
                        System.err.println("D:"+ e.getMessage());
                }
                else if ( e instanceof InvalidIndexException )
                        {
                                System.err.println("I:"+ e.getMessage());
                        }
        }
}
///////    TestMatrix.java    Fin ///////

///////    MatrixException.java  Inicio ///////
package matrix;

public class MatrixException extends Exception {

        public MatrixException() {}

        public MatrixException(String s) {
                super(s);
        }

        /* Super: palabra clave Java. Referencia al objeto anfitrión como
instancia de su superclase. Tiene la misma funcionalidad que this, pero
se utiliza para acceder miembros heredados. En este caso en especial,
dentro del cuerpo del constructor de la sub clase, está invocando el
constructor de la super clase.                                    */
}
///////    MatrixException.java  Fin ///////
```

```
///////    IncompatibleDimException.java  Inicio  ///////
package matrix;

public class IncompatibleDimException extends MatrixException
{
       public IncompatibleDimException() {}

       public IncompatibleDimException(String s) {
              super(s);
       }   // msg string
}
///////    IncompatibleDimException.java  Fin ///////

///////    InvalidIndexException.java   Inicio ///////
package matrix;

public class InvalidIndexException extends MatrixException
{
       public InvalidIndexException() {}

       public InvalidIndexException(String s) {
              super(s);
       }   // msg string
}
///////    InvalidIndexException.java   Fin ///////

///////    SingularException.java    Inicio ///////
package matrix;

public class SingularException extends MatrixException
{
       public SingularException() {}

       public SingularException(String s) {
              super(s);
       }   // msg string
}
///////    SingularException.java    Fin ///////


///////////  Compile.NOTE ////////
To compile go to .. and do

javac matrix/*.java

to run go to .. and do

java matrix.TestMatrix
```

11. ENTRADAS Y SALIDAS

En java, la entrada y salida se define en términos de streams (corrientes) de datos. Los streams se procesan secuencialmente, y los más comunes son la entrada estándar (teclado) y la salida estándar (pantalla).

El paquete java.io es el que define las clases que componen la entrada/salida de java. Si queremos usar las clases que tenemos definidas en ese paquete, deberemos incluir la declaración de importación en el comienzo del programa:

Import java.io.*;

En general , las clases se agrupan en pares: por cada clase de stram de entrada hay una clase de stream de salida. Un stream no tiene que estar asociado a un archivo comun:

puede ser un periférico (como el teclado), o un archivo remoto recuperado por la Web.

La clase InputStream

Es una clase abstracta, de la que derivan las demas clases de streams de lectura. Las clases interesantes de entrada se definen como subclases derivadas de ésta. Algunos de los métodos más utilizados:

int read() : Lee un byte, devuelve un valor de 0 a 255 o –1 si no hay mas datos.

int read(byte [] buffer) : Lee la cantidad de bytes que especifique buffer.length.

void close() cierra el stream.

La clase OutputStream

Es una clase abstracta que sirve de base para las demás clases de salidas de datos. El método básico es:

void write (int b) : Graba el parámetro entero b como byte.

void write (byte [] buf) : Graba un arreglod de bytes.

void flush(): Fuerza la escritura vaciando el buffer.

A partir de estas clases, el paquete java.io define un conjunto de clases, cada uno adaptado a datos distintos.

Clases FileInputStream y FileOutputStram

Estas dos clases de file streams (corrientes de archivo) derivan de las clases abstractas InputStream y OutputStream, con lo que opueden leer y escribir bytes o arreglos e bytes.

FileInputStream es la clase que permite leer streams de archivos. El constructor pide el nombre de un archivo, un objeto file o un objeto FileDescriptor. Al pasarle un nombre de archivo, estamos abriendo para leer el archivo indicado.

FileOututStream es la contrapartida de la anterior, se usa para generar archivos de salida.

Clase PrintStream

Esta clase es una clase de salida, que tiene varios métodos print y println. Cada uno con varios versiones, que aceptan parámetros de distintos tipos, como enteros, enteros largos, caracteres, arreglos de caracteres, realeas, objetos y cadenas de caracteres. Esta clase se usa generalmente en el objeto System.out, para producir la salida por pantalla.

Ejemplos de entrada de datos

```
import java.io.*;public class Entrada1 {
        public      static      void      main(String      args[])      throws
        FileNotFoundException,IOException {
                FileInputStream     fptr;int                  n;

                fptr = new FileInputStream("Ejemplo9.java");
                do {
                        n = fptr.read();
                        if (n!=-1) System.out.print((char)n);
                } while (n!=-1);
                fptr.close();
        }
}
```

Otro ejemplo, esta vez manejando excepciones

```
import java.io.*;

public class Entrada2 {
	public static void main(String args[]) {
		FileInputStream	fptr;
		DataInputStream	f;
		String		linea = null;

		try {
			fptr = new FileInputStream(args[0]);
			f = new DataInputStream(fptr);
			do {
				linea = f.readLine();
				if (linea!=null) System.out.println(linea);
			} while (linea != null);
			fptr.close();
		}
		catch (FileNotFoundException e) {
			System.out.println("ese archivo no existe!");
		}
		catch (IOException e) {
			System.out.println("Error de E/S!\n");
		}
	}
}
```

El archivo a imprimir se elige desde la línea de comandos, en lugar de entrarlo fijo, utilizando para eso el argumento del método main(arg[]), que consiste en una lista de Strings con los parámetros que se pasan en la línea a continuación de java nombre_programa. Por ejemplo, podríamos llamar a este programa con:

```
java Entrada2 archi.txt otro.xxx
```

arg[0] contendrá archi.txt, arg[1] contendrá otro.xxx, y así sucesivamente. Por supuesto, si llamamos a Entrada2 sin parámetros se lanzará otra excepción al intentar accederlo.

Clase In

A continuación vamos a presentar una clase creada por el Ing. Gustavo Garcia para facilitar el ingreso de datos en Java.

La clase In permite que una aplicación pueda leer datos de pantalla fácilmente. Ha sido desarrollada para tener un comportamiento "similar" al cin de C++.

utn.frc.io

Class In

```
java.lang.Object
  |
  +--utn.frc.io.In
```

public class In extends java.lang.Object

A diferencia de C++, en Java no existe la sobrecarga de operadores. Por ello, la única forma posible de leer distintos tipos de datos es ofreciendo un conjunto de métodos de entrada. Todos los métodos de entrada definidos en esta clase (con sólo una excepción) poseen el mismo formato:

* <tipo> read<Tipo>()

donde <tipo> representa el tipo de dato retornado, pudiendo éste ser char, int, long, float, double o String, y

<Tipo> es el mismo que fue indicado como tipo de retorno, pero comenzando en mayúsculas, o Line.

Todos los métodos son estáticos (static) por lo que no es necesario crear una instancia de esta clase para realizar la lectura de datos. Es más, para evitar la innecesaria instanciación de In su constructor se ha definido privado. Esto impide tanto la creación de objetos de esta clase como su derivación o extensión.

El siguiente es un ejemplo de uso de los métodos de entrada de esta clase. Si ejecutamos el main() que sigue:

```
import utn.frc.io.In;

public static void main(String[] args) {
   int i;
   long lng;
   float f;
   String str;
   String line;

   System.out.print("Introduzca un entero: ");
   i = In.readInt();
   System.out.print("Introduzca una cadena (sin espacios): ");
   str = In.readString();
   System.out.print("Introduzca un entero largo: ");
   lng = In.readLong();
   System.out.print("Introduzca una línea: ");
   line = In.readLine();
   System.out.print("Introduzca un float: ");
   f = In.readFloat();

   System.out.println();
   System.out.println("Entero leído: " + i);
   System.out.println("Entero largo leído: " + lng);
   System.out.println("Cadena leída: " + str);
   System.out.println("Línea leída: " + line);
   System.out.println("Float leído: " + f);
}
```

Podemos tener la siguiente ejecución por pantalla:

```
Introduzca un entero: 123
Introduzca una cadena (sin espacios): hola!
Introduzca un entero largo: 45678
Introduzca una línea: Hola Mundo!
Introduzca un float: 123.20

Entero leído: 123
Entero largo leído: 45678
```

```
Cadena leída: hola!
Línea leída: Hola Mundo!
Float leído: 123.20
```

Sin embargo, si realizamos una ejecución diferente, podremos verificar que los valores introducidos que no son utilizados por alguno de los métodos read<Tipo>() quedan disponibles para el siguiente read<Tipo>() que los pueda utilizar. Por ejemplo:

```
Introduzca un entero: 123 hola!
Introduzca una cadena (sin espacios): Introduzca un entero largo: 45678
Hola Mundo!
Introduzca una línea:
Entero leído: 123
Entero largo leído: 45678
Cadena leída: hola!
Línea leída: Hola Mundo!
Float leído: 123.20
```

12. LIBRERIAS ESTANDAR

Java viene con un conjunto muy completo de librerías. Estas tienen un valor añadido muy importante, porque las librerías son indispensables en un desarrollo llevado con técnicas orientadas a objetos.

Las librerías:

- disminuyen su carga de trabajo;
- estandarizan sus aplicaciones;
- le proporcionan ejemplos de código fuente Java de referencia.

Tomemos un ejemplo: la clase Object proporciona el método toString(). Este método es útil especialmente para depurar y usted puede/debe redefinirlo para las clases que cree. Pues bien, el método toString() de la clase Vector llama simplemente a los métodos toStringq() de los objetos que contiene el Vector. Así, el programa siguiente:

```
import java.util.Vector;
public class trace {
      public static void main (String [ ] arg) {
            Vector v = new Vector ();
            v.addElement (new AA (“UNO”));
            v.addElement (new AA (“DOS”));
            v.addElement (new AA (“TRES”));
            system.out.println (v);
      }
}

class AA {
      String nombre;
      public AA (String elNombre) {
            nombre = elNombre;
      }
      public String toString () {
            return nombre;
      }
}
```

tras la compilación y la ejecución obtendrá:

```
C:\ProgramasJava\libreriaa>java trace [UNO, DOS, TRES]
```

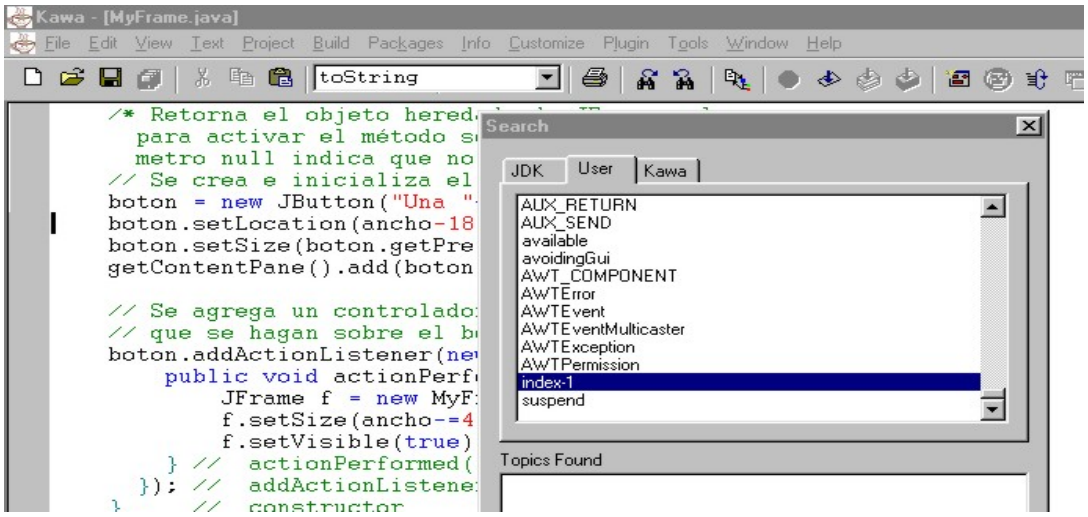
En Java hay muchas clases ya definidas y utilizables. Las que se presentan a continuación vienen en las bibliotecas estándar:

- java.lang: clases esenciales, números, strings, objetos, compilador, runtime, seguridad y *threads* (es el único paquete que se incluye automáticamente en todo programa java).
- java.io: clases que manejan entradas y salidas.
- java.util: clases útiles, como estructuras genéricas, manejo de fecha, hora y strings, números aleatorios, etc.
- java.net: clases para soportar redes: URL, TCP, UDP, IP, etc.
- java.awt: clases para manejo de interfase gráfica, ventanas, etc.
- java.awt.image: clases para manejo de imágenes.
- java.awt.peer: clases que conectan la interfase gráfica a implementaciones dependientes de la plataforma (motif, windows).
- java.applet: clases para la creación de applets y recursos para reproducción de audio.

13. AYUDA EN LINEA

Como Ud puede apreciar en los ejemplos comentarizados, Java incluye una biblioteca muy grande. Está organizada en paquetes conteniendo clases e interfaces. Las clases están organizadas en una estructura hereditaria simple. Cada una contiene desde unos pocos hasta centenares de métodos, y además tenemos herencia y sobrecarga. Para poder entender, más aún codificar un programa es imprescindible tener un eficiente acceso a esta información

Esta ayuda en línea se encuentra en un conjunto de archivos .HTML que se generan automaticamente en la instalación del JDK, en el directorio ... Java\jdk1.3\docs\api\index_files. Se puede acceder a ellos desde el explorador de windows, clickeando sobre index1.HTML. Si se está trabajando dentro de un entorno de desarrollo (aconsejado), por ejemplo Kawa, es conveniente referenciar esta documentación dentro del propio help de Kawa. Clickeando **Help / Search / User**:



Posicionamos index1, click:

Vemos la ventana inicial del Help. Si clickeamos sobre:

- **Overview** Página inicial de este documento API. Proporciona una lista de los paquetes, con un breve sumario de cada uno.

- **Package** Cada paquete contiene una página con una lista de sus clases e interfaces, incluyendo un sumario para cada una.

- **Clase / Interfase** Cada una contiene una página. Cada página contiene secciones:
  - Descripción
    - Diagrama de herencia
    - Subclases nivel inmediato inferior.
    - Todas las subinterfaces conocidas
    - Todas las clases de implementación
    - Su declaración
    - Su descripción
  - Sumario de tablas
    - Sumario de clases derivadas
    - Sumario de campos
    - Sumario de constructores
    - Sumario de métodos
  - Descripción detallada
    - Campos
    - Costructores
    - Métodos

- **Use** Es una referencia cruzada. Cada paquete, clase e interfase posee una página. Dicha página describe que paquetes, clase, métodos, constructores y campos usan cualquier parte de ella. Se puede acceder a Use estando posicionado en un paquete, clase o interface.

- **Tree** (Jerarquía de clases) Existe una jerarquía a nivel de paquetes y otra dentro de cada paquete. Las clases son organizadas en una estructura de herencia partiendo de java.lang.Object. Las interfaces no heredan desde java.lang.Object.

- **Deprecated API** Esta página contiene todas las API obsoletas. No deben ser usadas en aplicaciones actuales, solo se documentan a los efectos de la compatibilidad hacia atrás.

- **Index** Indice alfabético de todo: clases, interfaces, constructores, métodos, campos.

- **Prev / Next** Permite lectura secuencial en avance/retroceso de la documentación

PRESENTACION DE CODIGO CON UML

INTRODUCCION

UML (Unified Modeling Language, lenguaje unificado de modelado) es una anotación estándar para el modelado de sistemas orientados a objetos. UML es, básicamente, un lenguaje que describe gráficamente un conjunto de elementos. Sus usos más complejos son los de especificar, presentar, construir y documentar sistemas de software y de otros tipos, así como modelos empresariales. Como los planos de construcción de los edificios, UML proporciona una representación gráfica del diseño de un sistema, que puede ser esencial para la comunicación entre los miembros del equipo y para garantizar la coherencia arquitectónica del sistema.

En la presentación del código, UML constituye una útil herramienta de examen, análisis del desarrollo de las aplicaciones y comunicación del diseño del software.

Los diagramas UML pueden ayudarle a comprender rápidamente la estructura de un código desconocido, reconocer áreas especialmente complejas, e incrementar su productividad resolviendo problemas con mayor rapidez.

JAVA y UML

Dado que Java y UML son lenguajes orientados a objetos e independientes de la plataforma, se integran sin problemas. UML, una valiosa herramienta para la comprensión de Java y las complejas relaciones entre clases y paquetes, ayuda a los desarrolladores a captar el significado de las clases y el paquete completo en el que se encuentran. Sobre todo, puede ayudar a los desarrolladores en Java que se incorporan a un equipo, a familiarizarse rápidamente con la estructura y el diseño del sistema de software.

El objetivo de la materia es construir, en base a un modelo de sistema, un programa en un lenguaje de programación orientado a objetos(Java). Es decir, que nuestra tarea no consistirá en realizar un modelo de la realidad (objetivos de materias posteriores en la carrera) sin tomar un modelo representado con a herramienta UML e implementarlo en un lenguaje concreto como Java.

TERMINOS DE JAVA Y UML

Dado que UML está concebido para describir muchos entornos distintos, utiliza términos genéricos para referirse a las relaciones. En la tabla siguiente se definen los términos propios de Java y sus equivalentes en UML. En algunos casos, los términos son idénticos. En esta documentación se utilizan los correspondientes a Java.

Término Java	Definición Java	Términos de UML	Definición UML
Herencia	Mecanismo por el cual una clase o interfaz se define como especialización de otra más amplia. Las clases e interfaces herederas utilizan la palabra clave extends .	Generalización / especialización	Una relación entre un elemento especializado y otro generalizado, en la que el primero incorpora la estructura y el comportamiento del segundo.
Dependencia	Relación en la que los cambios realizados en un objeto independiente pueden influir sobre otro que depende de él.	Dependencia	Relación en la que las características semánticas de una entidad dependen de las de otra y están determinadas por ella.

Asociación	Dependencia especializada en la que se almacena una referencia a otra clase.	Relación (asociación)	Relación de estructura que describe los vínculos entre objetos.
Interfaz	Grupo de constantes y declaraciones de métodos que definen la forma de una clase sin implementar los métodos. En la interfaz se establece lo que debe hacer una clase, pero no la forma en que debe hacerlo. Las clases e interfaces que implementan la interfaz utilizan la palabra clave implements .	Realización / interfaz	Conjunto de operaciones que especifican un servicio de una clase o componente. Definen el comportamiento de una abstracción sin implementarlo.
Método	Implementación de una operación definida por una interfaz o una clase.	Operación	Implementación de un servicio que puede solicitar un objeto y puede influir en su comportamiento. Normalmente, en los diagramas UML las operaciones se enumeran debajo de los atributos.
Campo	Variable de instancia o miembro de datos de un objeto.	Atributo	Propiedad de un clasificador, como una clase o una interfaz, que describe los valores que pueden adoptar las instancias de una propiedad.
Propiedad	Información sobre el estado actual de un componente. Las propiedades de los componentes pueden considerarse atributos que tienen un nombre concreto y que puede leer (get), definir (set) o poner (put) un usuario o un programa.		En los diagramas UML existe una propiedad cuando el nombre de un campo coincide con el de un método, este último precedido por “is”, “set”, “get” o “put”. Por ejemplo, un campo llamado parameterRow es una propiedad si tiene un método llamado setParameterRow().

COMPONENTES DE LOS DIAGRAMAS UML

En la tabla siguiente se definen las carpetas del panel de estructura y los términos que aparecen en los diagramas. También se indica su representación en UML.

Término o Elemento	**Definición**	**Representación**	**Ejemplo de diagrama**
Clases Base	Clases de las que otra clase hereda los atributos (campos y propiedades) y métodos.	Una línea continua con un gran triángulo que apunta de la subclase a la superclase.	
Clases	Objetos que definen objetos. Las definiciones de clases definen campos y métodos.	Se muestran en un recuadro rectangular, con el nombre en la parte superior y los campos, métodos y propiedades por debajo.	Prueba + main(arg: String[]): void

Clases abstractas	Clases de las que no se pueden crear instancias.	Se muestran en cursiva.	*Prueba* + main(arg: String[]): void
Clases herederas	Clases que extienden la superclase	Una línea continua con un gran triángulo que apunta de la subclase a la superclase.	
Interfaces	Grupos de constantes y declaraciones de métodos que definen la forma de una clase sin implementar los métodos. Las interfaces permiten establecer lo que debe hacer una clase, pero no la forma en que debe hacerlo.	Un rectángulo relleno. El nombre de la interfaz se presenta en cursiva.	*Prueba*
Relación de uso	Dependencias especializadas en las que se almacena una referencia a otra clase.	Una línea punteada acabada en punta de flecha.	
Métodos	Operaciones definidas en una clase o una interfaz.	Se enumeran bajo los nombres y los campos, e incluyen el tipo de devolución y los parámetros.	Mostrar():void
Métodos abstractos	Métodos sin implementación.	Se muestran en cursiva.	*getSueldo(): float*
Miembros / campos	Variables de instancia o miembros de datos de un objeto.	Se enumeran bajo el nombre de la clase.	conterPanel: JPanel
Estático	Con ámbito de clase.	Los miembros, campos, variables y métodos estáticos se muestran subrayados en el diagrama UML.	numero: int

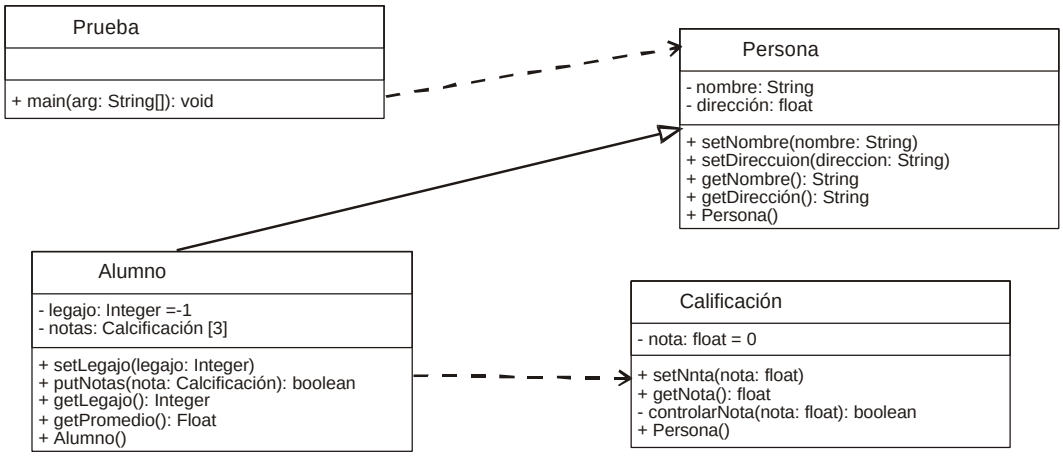
ICONOS DE ACCESIBILIDAD

En UML se utilizan iconos para representar la accesibilidad de las clases, como public, private, protected y package .

En UML la accesibilidad de las clases se representa por medio de iconos más genéricos, definidos en la tabla siguiente.

Icono UML	**Descripción**
+	public
-	private
#	protected

A continuación, presentamos, a modo de ejemplo, un diagrama combinado de clases



EJERCICIOS PRACTICOS

EJERCICIOS RESUELTOS

Los siguientes ejercicios definen, implementan y utilizan distintos tipos de clases y situaciones.

1. Clase PC.

```
//archivo: PC.java

public class PC {

  String  Marca;
  String  Procesador;
  String  Pantalla;
  boolean PCEncendido;

  public void Inicializar(String marca, String procesador,
                          String pantalla) {
      Marca = marca;
      Procesador = procesador;
      Pantalla = pantalla;
      PCEncendido = false;
  }

  public void EncenderPC()  {
    if (PCEncendido == true) // si está encendido...
              System.out.println("El ordenador ya está encendido.");
    else // si no está encendido, encenderlo.
           {
            PCEncendido = true;
            System.out.println("El ordenador se ha encendido.");
           }
  }

  public void Estado()  {
    System.out.println("\nEstado del ordenador:" +
                       "\nMarca " + Marca +
                       "\nProcesador " + Procesador +
                       "\nPantalla " + Pantalla + "\n");
    if (PCEncendido == true) // si el ordenador está encendido...
              System.out.println("El ordenador está encendido.");
    else // si no está encendido...
              System.out.println("El ordenador está apagado.");
  }
}

//archivo: MiPC.java

import java.io.*;
import utn.frc.io.In;

public class MiPC
{
  public static void main (String[] args)
  {
    PC MiPC = new PC();
    MiPC.Inicializar("Toshiba", "Intel Pentium", "TFT");
    MiPC.EncenderPC();
    MiPC.Estado();
  }
}
```

2. Clase Cuenta bancaria, manejo de vectores de objetos.

```
//archivo: CuentaBancaria.java

public class CuentaBancaria{
       private float balance;

       public void Inicializar(float balance){
              this.balance=balance;
       }
       public void Depositar(float cantidad){
              balance+=cantidad;
       }
       public boolean Retirar(float cantidad){
              if(balance>=cantidad){
                     balance-=cantidad;
                     return true;
              }
              return false;
       }
       public float ObtenerBalance(){
              return balance;
       }
}

//Archivo: CuentaApp.java

import java.io.*;
import utn.frc.io.In;

public class CuentaApp {
    public static void main(String[] args) {
        //Se crea el objeto cuenta
      CuentaBancaria cuenta = new CuentaBancaria ();

      //Inicializa
      cuenta.Inicializar(0);

      //Deposita
      cuenta.Depositar(100);

      //retirar
      if (cuenta.Retirar(50))
        System.out.println("Se retiró exitosamente");
      else
        System.out.println("No hay fondos para retirar");

      System.out.println("El balance es : "+ cuenta.ObtenerBalance());

      //usando la entrada in
       CuentaBancaria cuenta2 = new CuentaBancaria ();

      //Inicializar
      System.out.print("Ingrese el importe que inicializará la cuenta :");
      float importe = In.readFloat();
      cuenta2.Inicializar(importe);

      //Depositar
      System.out.print("Ingrese el importe a depositar :");
      importe = In.readFloat();
      cuenta2.Depositar(importe);

      //retirar
      System.out.print("Ingrese el importe a retirar :");
      importe = In.readFloat();
      if (cuenta2.Retirar(importe))
        System.out.println("Se retiró exitosamente");
      else
        System.out.println("No hay fondos para retirar");
```

```
        //Mostrar el saldo
        System.out.println("El balance es : "+ cuenta2.ObtenerBalance());

        //Usando un vector de objetos
        CuentaBancaria cuentas[] = new CuentaBancaria [10];
        int i;

        for (i=0 ; i<10 ; i++)
              cuentas[i] = new CuentaBancaria ();

        //Inicializar
        for (i=0 ; i<10 ; i++) {
             System.out.print("Ingrese el importe que inicializará la cuenta: ");
             importe = In.readFloat();
             cuentas[i].Inicializar(importe);
        }

        //Depositar
        for (i=0 ; i<10 ; i++) {
             System.out.print("Ingrese el importe a depositar :");
             importe = In.readLong();
             cuentas[i].Depositar(importe);
        }

        //retirar
        for (i=0 ; i<10 ; i++) {
          System.out.print("Ingrese el importe a retirar :");
          importe = In.readLong();
          if (cuentas[i].Retirar(importe))
            System.out.println("Se retiró exitosamente ");
          else
            System.out.println("No hay fondos para retirar");
        }

        //Mostrar el saldo
        for (i=0 ; i<10 ; i++)
          System.out.println("El saldo es : "+ cuentas[i].ObtenerBalance());
      }
}
```

3. Clase reloj, utilización de constructores.

```
//archivo: Reloj.java
public class Reloj {
       private int horas, minutos, segundos;

      //constructor UNO
       public Reloj() {
          horas = 12;
          minutos = 0;
          segundos = 0;
       }

       //constructor DOS
       public Reloj(int h, int m, int s) {
          horas = h;
          minutos = m;
          segundos = s;
       }

      //funcion pública UNO
       public void cambiarHora(int h, int m, int s) {
          horas = h;
          minutos = m;
          segundos = s;
       }

       //funcion pública DOS
```

```
        public void cambiarHora(int h) {
            horas = h;
         }

         //otra función pública
         public String obtenerHora() {
            return "Hora: "+horas+" Minutos: "+minutos+" Segundos: "+segundos;
         }

         public void incrementarHora() {
            segundos++;
            if (segundos == 60)
            {
               segundos = 0;
               minutos++;
               if (minutos == 60)
               {
                      minutos = 0;
                      horas++;
                      if (horas == 24)
                      {
                             horas = 0;
                             minutos = 0;
                             segundos = 0;
                      }
                }
            }
         }
}


//archivo: RelojApp.java

import java.io.*;
import utn.frc.io.In;

public class RelojApp {
    public static void main(String[] args) {
      Reloj r1,r2;
      r1 = new Reloj();
      r2 = new Reloj(12,25,30);

      r1.cambiarHora(15);
      System.out.println("La hora es:  "+r1.obtenerHora());
      r1.incrementarHora();
      System.out.println("La hora es:  "+r1.obtenerHora());

      r1.cambiarHora(15,20,30);
      System.out.println("La hora es:  "+r1.obtenerHora());
      r1.incrementarHora();
      System.out.println("La hora es:  "+r1.obtenerHora());
    }
}
```

4. Clase Persona, utilización de constructores y vectores de objetos.

```
//archivo: Persona.java

public class Persona {
  private String Nombre;
  private String Domicilio;
  private String Telefono;
  private String DNI;
  private int Edad;

  public Persona ()       //constructor sin argumentos
  {
         Nombre = "";
```

```
        Domicilio = "";
        Telefono = "";
        DNI = "";
        Edad  = 0;
    }

    public Persona ( String nom, String dom, String tel, String dni, int edad) {
        Nombre = nom;
        Domicilio = dom;
        Telefono = tel;
        DNI = dni;
        Edad  = edad;
    }

    public void SetNombre (String nom)   { Nombre = nom; }
    public void SetDomicilio (String dom) { Domicilio = dom; }
    public void SetTelefono (String tel)  { Telefono = tel; }
    public void SetDni (String dni)      { DNI = dni; }
    public void SetEdad (int edad)       { Edad = edad; }
    //funciones que muestran el valor de los datos miembros
    public String GetNombre ()           { return Nombre; }
    public String GetDomicilio ()        { return Domicilio; }
    public String GetTelefono ()         { return Telefono; }
    public String GetDNI ()              { return DNI; }
    public int  GetEdad ()               { return Edad; }
}


//archivo: PersonaApp.java
import java.io.*;
import utn.frc.io.In;

public class PersonaApp {
    public static void main(String[] args) {
       int i;
       //Se crea el objeto p1 llamando al constructor sin argumentos
       Persona p1 = new Persona();
       //Se crea el objeto p2 llamando al constructor con argumentos
       Persona p2=new Persona("Juan Perez","La Paz 134","643552", "22.222.222",
                                  20);
       //Se crea un vector de objetos tipo persona
       Persona p3[] = new Persona[2];
       for (i=0 ; i<2 ; i++)
              p3[i] = new Persona();

       //cargar los objetos
       p1.SetNombre("María");
       p1.SetDomicilio("Colon 234");
       p1.SetTelefono("453256");
       p1.SetDni("23.333.444");
       p1.SetEdad(23);

       for (i=0 ; i<2 ; i++) {
         String n, d, t, dni;
         int edad;

         System.out.print("Introduzca el nombre: ");
         n = In.readLine();
         System.out.print("Introduzca el domicilio: ");
         d = In.readLine();
         System.out.print("Introduzca el telefono: ");
         t = In.readLine();
         System.out.print("Introduzca el dni: ");
         dni = In.readLine();
         System.out.print("Introduzca la edad: ");
         edad = In.readInt();

         p3[i].SetNombre(n);
```

```
            p3[i].SetDomicilio(d);
            p3[i].SetTelefono(t);
            p3[i].SetDni(dni);
            p3[i].SetEdad(edad);
          }
          System.out.println("Nombre    Domicilio   Telefono    DNI   Edad ");

          //mostrar el objeto p1
          System.out.println(p1.GetNombre() + " " + p1.GetDomicilio()+" " +
          p1.GetTelefono()+ " " + p1.GetDNI()+ " " + p1.GetEdad());

          //mostrar el objeto p2
          System.out.println(p2.GetNombre() + " " + p2.GetDomicilio()+" " +
                p2.GetTelefono()+ " " + p2.GetDNI()+ " " + p2.GetEdad());

          //Mostrar el vector
          for (i=0 ; i<2 ; i++) {
            System.out.println(p3[i].GetNombre() + " " + p3[i].GetDomicilio()+" " +
                p3[i].GetTelefono()+ " " + p3[i].GetDNI()+ " " + p3[i].GetEdad());
          }
      }
  }
```

5. Clase rectangulo

Vamos a crear una clase denominada Rectangulo, que describa las características comunes a estas figuras planas.

Atributos:

x e y : Representan el origen del rectángulo, posición de la esquina superior izquierda del rectángulo en el plano.

ancho y alto: Representan las dimensiones del rectángulo.

Métodos:

Constructor1: Crea un rectángulo de dimensiones nulas situado en el punto (0, 0),

Constructor2: Recibe cuatro números que guardan los parámetros x1, y1, w y h, y con ellos inicializa los miembros dato x, y, ancho y alto.

Constructor3: Crea un rectángulo cuyo origen está en el punto (0, 0).

CalcularArea: Función que calcula el área realizará la siguiente tarea, calculará el producto del ancho por el alto del rectángulo y devolverá el resultado. La función devuelve un entero es por tanto, de tipo int. No es necasario pasarle datos ya que tiene acceso a los miembros dato ancho y alto que guardan la anchura y la altura de un rectángulo concreto.

Desplazar: función que desplaza el rectángulo horizontalmente en dx, y verticalmente en dy, le pasamos dichos desplazamientos, y a partir de estos datos actualizará los valores que guardan sus miembros dato x e y.

EstaDentro: Función que determina si un punto está o no en el interior del rectángulo, devolverá true si el punto se encuentra en el interior del rectángulo y devolverá false si no se encuentra, es decir, será una función del tipo boolean. La función necesitará conocer las coordenadas de dicho punto. Para que un punto de coordenadas x1 e y1 esté dentro de un rectángulo cuyo origen es x e y, y cuyas dimensiones son ancho y alto, se deberá cumplir a la vez cuatro condiciones

x1>x y a la vez x1<x+ancho

También se debe cumplir

y1>y y a la vez y1<y+alto

Como se tienen que cumplir las cuatro condiciones a la vez, se unen mediante el operador lógico AND simbolizado por &&.

Se escribe la clase, y se guarda en un archivo que tenga el mismo nombre que la clase Rectangulo y con extensión .java.

```
//archivo: Rectangulo.java
public class Rectangulo {
    int x;
    int y;
    int ancho;
    int alto;
    public Rectangulo() {
        x=0;
        y=0;
        ancho=0;
        alto=0;
    }
    public Rectangulo(int x1, int y1, int w, int h) {
        x=x1;
        y=y1;
        ancho=w;
        alto=h;
    }
    public Rectangulo(int w, int h) {
        x=0;
        y=0;
        ancho=w;
        alto=h;
    }
    int calcularArea(){
       return (ancho*alto);
   }
   void desplazar(int dx, int dy){
       x+=dx;
       y+=dy;
   }
   boolean estaDentro(int x1, int y1){
       if((x1>x)&&(x1<x+ancho)&&(y1>y)&&(y1<y+ancho))
                  return true;
       return false;
   }
}

//archivo: RectanguloApp.java
import java.io.*;
import utn.frc.io.In;

public class RectanguloApp {
     public static void main(String[] args) {
       Rectangulo rect1=new Rectangulo(10, 20, 40, 80);
       Rectangulo rect2=new Rectangulo(40, 80);
       Rectangulo rect3=new Rectangulo();

       int medidaArea=rect1.calcularArea();

       System.out.println("El área del rectángulo es "+medidaArea);

       rect2.desplazar(10, 20);

       if(rect1.estaDentro(20,30))
               System.out.println("El punto está dentro del rectángulo");
       else
               System.out.println("El punto está fuera del rectángulo");
    }
}
```

6. La clase Punto y La clase Rectangulo

Hay dos formas de reutilizar el código, mediante la composición y mediante la herencia. La composición significa utlizar objetos dentro de otros objetos. Por ejemplo, un applet es un objeto que contiene en su interior otros objetos como botones, etiquetas, etc. Cada uno de los controles está descrito por una clase.

Vamos a estudiar una nueva aproximación a la clase Rectangulo definiendo el origen, no como un par de coordenadas x e y (números enteros) sino como objetos de una nueva clase denominada Punto.

La clase Punto

Atributos:

La abscisa x y la ordenada y de un punto del plano.

Métodos:

Constructor1: Constructor por defecto que sitúa el punto en el origen.

Constructor2: Constructor explícito que proporciona las coordenadas x e y de un punto concreto.

Desplazar: Cambia la posición del punto desde (x, y) a (x+dx, y+dy). La función desplazar cuando es llamada recibe en sus dos parámetros dx y dy el desplazamiento del punto y actualiza las coordenadas x e y del punto. La función no retorna ningún valor

```
//archivo: Punto.java
public class Punto {
  int x = 0;
  int y = 0;
  public Punto(int x, int y) {
        this.x = x;
       this.y = y;
  }
  public Punto() {
       x=0;
       y=0;
  }
  void desplazar(int dx, int dy){
       x+=dx;
       y+=dy;
  }
}
```

La clase Rectangulo

Atributos:

La clase Rectangulo tiene como atributos, el origen que es un objeto de la clase Punto y las dimensiones ancho y alto.

Metodos:

Constructor1: Constructor por defecto, crea un rectángulo situado en el punto 0,0 y con dimensiones nulas.

Constructor2: Constructor explícito crea un rectángulo situado en un determinado punto p y con unas dimensiones que se le pasan en el constructor

Desplazar: Desplaza un rectángulo, trasladamos su origen (esquina superior izquierda) a otra posición, sin cambiar su anchura o altura.

```
//archivo: Rectangulo.java
public class Rectangulo {
```

```
        Punto origen;
        int ancho ;
        int alto ;

        public Rectangulo() {
            origen = new Punto(0, 0);
            ancho=0;
            alto=0;
        }

        public Rectangulo(Punto p, int w, int h) {
            origen = p;
            ancho = w;
            alto = h;
        }

        void desplazar(int dx, int dy) {
            origen.desplazar(dx, dy);
        }

        int calcularArea() {
            return ancho * alto;
        }
    }
```

Manipulación de Objetos de la clase Rectangulo

```
    //archivo: RectanguloApp.java

    import java.io.*;
    import utn.frc.io.In;

    public class RectanguloApp {
        public static void main(String[] args) {
            Punto p = new Punto(10,15);
            Rectangulo rect1=new Rectangulo(p, 100, 200);
            Rectangulo rect2=new Rectangulo(new Punto(44, 70), 10,20);
            Rectangulo rect3=new Rectangulo();
            rect1.desplazar(40, 20);
            System.out.println("el área es "+rect1.calcularArea());
            int areaRect;

            areaRect = rect2.calcularArea();
            System.out.println("el área es "+areaRect);
        }
    }
```

7. Clase vector, maneja un vector de enteros

```
    //archivo: Vector.java
    public class Vector {
        int[] x;     //vector
        int n;      //dimensión
        public Vector(int[] x) {
            this.x=x;
            n=x.length;
        }
        double valorMedio(){
            int suma=0;
            for(int i=0; i<n; i++){
                suma+=x[i];
            }
            return (double)suma/n;
        }
        int valorMayor(){
            int mayor=x[0];
            for(int i=1; i<n; i++){
                if(x[i]>mayor)  mayor=x[i];
            }
```

```
            return  mayor;
        }
        int valorMenor(){
            int menor=x[0];
            for(int i=1; i<n; i++){
                if(x[i]<menor)  menor=x[i];
            }
            return  menor;
        }
        void ordenar(){
            int aux;
            for(int i=0; i<n-1; i++){
                for(int j=i+1; j<n; j++){
                    if(x[i]>x[j]){
                        aux=x[j];
                        x[j]=x[i];
                        x[i]=aux;
                    }
                }
            }
        }
        void imprimir(){
            ordenar();
            for(int i=0; i<n; i++){
                System.out.print("\t"+x[i]);
            }
            System.out.println("");
        }
    }

    //archivo: VectorApp.java

    import java.io.*;
    import utn.frc.io.In;

    public class VectorApp {
        public static void main(String[] args) {
          int[] valores={10, -4, 23, 12, 16};
          Vector lista=new Vector(valores);
          System.out.println("Valor mayor "+lista.valorMayor());
          System.out.println("Valor menor "+lista.valorMenor());
          System.out.println("Valor medio "+lista.valorMedio());
          lista.imprimir();
        }
    }
```

8. Clase List, maneja una lista enlazada.

```
    //archivo: ListNode.java
    //una clase para los nodos de la lista

    class ListNode {
      Object data;                    //Dato a guardar
      ListNode next;           //Puntero al siguiente elemento
      //Constructor: para el nodo final de la lista
      ListNode(Object obj){
            data = obj;
        next = null;           //No hay ninguno próximo
      }
      //Constructor: Para nodos intermedios o iniciales
      ListNode(Object obj, ListNode nextNode){
            data = obj;
        next = nextNode;
      }
      //Retornamos el obj de este nodo
      Object getObject (){
        return data;
      }
```

```
  //Retornamos el siguiente nodo
  ListNode getNext() {
    return next;
      }
} //Fin de la clase ListNode


//archivo: List.java
//Definición de la clase lista

public class List {
  private ListNode firstNode;
  private ListNode lastNode;
  private String name;           //nombre de la lista
  //Construimos una lista vacía
  public List(String s){
      name = s;
      firstNode = lastNode = null;
  }

  //Otro constructor
  public List() {
    this("lista");
  }
  //Insertamos un objeto en la cabeza de la lista.
  //Si la lista está vacía, firstNode y lastNode se refieren al
  //mismo objeto. De otra manera, a firstNode le asignamos el
  //nuevo nodo.
  public void insertAtFront(Object insertItem){
    if ( isEmpty() )
      firstNode = lastNode = new ListNode(insertItem);
    else
      firstNode = new ListNode(insertItem, firstNode);
  }
  //Insertamos un objeto en la cola de la lista.
  //Si la lista está vacía, firstNode y lastNode se refieren al
  //mismo objeto. De otra manera, el handle next de lastNode
  //apuntará al objeto introducido.
  public void insertAtBack( Object insertItem ){
    if ( isEmpty() )
       firstNode = lastNode = new ListNode( insertItem);
    else
      lastNode=lastNode.next= new ListNode(insertItem);
  }
  //Eliminamos el primer nodo de la lista.
  public Object removeFromFron() throws EmptyListException {
    Object removeItem = null;
    //Si está vacía arrojamos una excepción
    if ( isEmpty() )
      throw new EmptyListException(name);
    removeItem = firstNode.data;
    //Si sólo existe un elemento
    if ( firstNode.equals( lastNode ) )
      firstNode = lastNode = null;
    else
      firstNode = firstNode.next;
    return removeItem;
  }
  //Eliminamos el último nodo de la lista
  public Object removeFromBack() throws EmptyListException {
            Object removeItem = null;
    //Si está vacía arrojamos un execpción
    if ( isEmpty() )
      throw new EmptyListException(name);
    removeItem = lastNode.data;
    //Si sólo existe un nodo
    if ( firstNode.equals(lastNode))
        firstNode = lastNode = null;
```

```
      else
      {
          ListNode current = firstNode;
          while (current.next != lastNode)
               current = current.next;
          lastNode = current;
          current.next = null;
      }
      return removeItem;
    }
    //¿Está vacía la lista?
    public boolean isEmpty() {
      return firstNode == null;
    }
    //Impresión de los contenidos de la lista
    public void print() {
      if ( isEmpty()){
        System.out.println( name+" vacía");
        return;
      }
      System.out.println("Los datos de "+ name +" son: ");
      ListNode current = firstNode;
      while ( current != null) {
        System.out.print(current.data.toString()+" ");
        current = current.next;
      }
      System.out.println();
    }
  }

  //archivo: EmptyListException.java
  //Definición de una clase que maneja excepcion

  class EmptyListException extends RuntimeException {
    public EmptyListException( String name) {
        super( "La "+ name + " está vacía");
    }
  }

  //archivo: ListApp.java

  import java.io.*;
  import utn.frc.io.In;

  public class ListApp {
      public static void main(String[] args) {
        List lista=new List("Lista 1");

        //inserta 5 elementos String al principio de la lista
        int i;
        String a;
        for (i=0; i<5; i++) {
          System.out.print("Introduzca un nombre: ");
          a = In.readLine();
         lista.insertAtFront(a);
        }

        //imprime
        lista.print();

        //borra los elementos
        for (i=0; i<5; i++) {
            a = lista.removeFromFron().toString();
            System.out.println("El nombre retirado es: "+a);
        }
    }
  }
```

9. Clase base Persona y clase derivada empleado. Implemementa herencia simple.

```
//archivo: Persona.java


public class Persona {

  protected String Nombre;

  protected String Domicilio;

  protected String Telefono;

  protected String DNI;

  protected int Edad;



public Persona ()         //constructor sin argumentos

{

       Nombre = "";
       Domicilio = "";
       Telefono = "";
       DNI = "";
       Edad  = 0;
}

public Persona ( String nom, String dom, String tel, String dni, int edad) {
       Nombre = nom;
       Domicilio = dom;
       Telefono = tel;
       DNI = dni;
       Edad  = edad;
}

public void SetPersona (String nom, String dom, String tel, String dni,
                           int edad)
{
       Nombre = nom;
       Domicilio = dom;
       Telefono = tel;
       DNI = dni;
       Edad = edad;
}

public String GetPersona ()
{
       String datos = Nombre+ " " +Domicilio+ " " +Telefono+" "+DNI+" "+Edad;
       return datos;//retorna una cadena con todos los datos
}
}//fin clase persona


//archivo: Empleado.java


public class Empleado extends Persona {

  protected int Legajo;

  protected String Cargo;



  public Empleado ()
```

```
  {

        super();
        Legajo = 0;
        Cargo = "";
  }

public Empleado ( String nom, String dom, String tel, String dni, int edad,
                    int leg, String cargo)
{
      super(nom, dom, tel, dni, edad);
      Legajo = leg;
      Cargo = cargo;
}

public void SetEmpleado (String nom, String dom, String tel, String dni,
                         int edad, int leg, String cargo)
{
      //accede directamente a los atributos de persona, por ser protegidos
      Nombre = nom;
      Domicilio = dom;
      Telefono = tel;
      DNI = dni;
      Edad = edad;
      //o bien llama al metodo publico SetPersona
      //SetPersona(nom, dom, tel, deni, edad);

      //carga los datos de empleado
      Legajo = leg;
      Cargo = cargo;
}

public String GetEmpleado ()
{
      //accede directamente a los atributos de persona, por ser protegidos
      String datos = Nombre + " " +    Domicilio + " " + Telefono + " " +
                         DNI+ " " + Edad + " " + Legajo + " " + Cargo;

      //o bien llama al metodo publico GetPersona
      //String datos = GetPersona()+ " " + Legajo + " " + Cargo;

      return datos;//retorna una cadena con todos los datos
}
}//fin clase Empleado


//archivo: EmpleadoApp.java


import java.io.*;

import utn.frc.io.In;



public class EmpleadoApp {

  public static void main(String[] args) {

    int i;
    Empleado p1 = new Empleado();
    //Se crea el objeto p2 llamando al constructor con argumentos
    Empleado p2 = new Empleado("Juan Perez", "La Paz 1234", "643552",
        "22.222.222", 20, 1, "gerente");
    //Se crea un vector de objetos tipo empleado
    Empleado p3[] = new Empleado[2];
    for (i=0 ; i<2 ; i++)
      p3[i] = new Empleado();
```

```
    //cargar los objetos
    p1.SetEmpleado("María", "Colon 234", "453256", "23.333.444", 23, 2,
                           "vendedor");

    for (i=0 ; i<2 ; i++) {
      String n, d, t, dni, c;
      int edad, leg;

      System.out.print("Introduzca el nombre: ");
      n = In.readLine();
      System.out.print("Introduzca el domicilio: ");
      d = In.readLine();
      System.out.print("Introduzca el telefono: ");
      t = In.readLine();
      System.out.print("Introduzca el dni: ");
      dni = In.readLine();
      System.out.print("Introduzca la edad: ");
      edad = In.readInt();
      System.out.print("Introduzca el legajo: ");
      leg = In.readInt();
      System.out.print("Introduzca el cargo: ");
      c = In.readLine();

      p3[i].SetEmpleado(n, d, t, dni, edad, leg, c);
       }
    System.out.println("Nombre    Domicilio   Telefono    DNI   Edad ");

    //mostrar el objeto p1
    System.out.println(p1.GetEmpleado());

    //mostrar el objeto p2
    System.out.println(p2.GetEmpleado());

    //Mostrar el vector
    for (i=0 ; i<2 ; i++) {
        System.out.println(p3[i].GetEmpleado());
       }
  }
}
```

10. Implementación de un listado de stock de prendas de vestir. Se utiliza un vector.

```
//archivo: Articulo.java


public class Articulo  {
   protected int Codigo;
   protected String Nom;
   protected int Cant;
   protected float Precio;

   public void SetArticulo(int cod, String nom, int can, float pre) {
        Codigo = cod;
        Nom = nom;
        Cant = can;
        Precio = pre;
   }

   public String GetArticulo() {
       return (Codigo + " " + Nom + " " + Cant + " " + Precio);
   }

   public int GetCodigo() { return Codigo;  }

   public String GetNombre() { return Nom; }

   public int GetCantidad() { return Cant; }
```

```
   public float GetPrecio() { return Precio; }

   public float TotalArt () { return (Cant * Precio); }

};

//archivo: Ropa.java

public class Ropa extends Articulo {
    protected int Talle;
    protected String Modelo;
    protected String Color;

    public void SetRopa(int cod, String nom, int can, float pre, int talle,
                        String mod, String col)
    {
        System.out.print("set Articulo ");
        SetArticulo(cod, nom, can, pre);
        System.out.print("set ropa ");
        Talle = talle;
        Modelo = mod;
        Color = col;
    }

    public String GetRopa() {
       return (GetArticulo() + " " + Talle + " " + Modelo + " " + Color);
    }

    public int GetTalle() { return Talle; }

    public String GetModelo() { return Modelo; }

    public String GetColor() { return Color; }
};


//archivo: LStock.java
import utn.frc.io.In;

public class LStock {
   private String titulo;
   private Ropa items[];
   private int elem;

   public LStock (int n, String t)
   {
        titulo = t;
        elem = n;
        items = new Ropa [elem];
        for (int i=0; i<elem; i++)
             items[i] = new Ropa();
   }

   public int TotalCant () {
       int sum =0;
       for (int i=0; i<elem; i++)
              sum += items[i].GetCantidad ();
        return sum;
   }

   float TotalPrecio () {
        float sum = 0;
       for (int i=0; i<elem; i++)
               sum += items[i].GetPrecio ();
        return sum;
   }
```

```
   float TotalGeneral () {
        float sum = 0;
       for (int i=0; i<elem; i++)
              sum += items[i].TotalArt();      //retorna cantidad * precio
        return sum;
   }

   void CargarLista ()
   {
       int cod, talle, can;
        String nom, mod, col;
        float pre;
        System.out.println("Ingrese los datos de la lista");
        for (int i=0; i<elem; i++)
        {
                System.out.print("Introduzca el codigo: ");
                cod = In.readInt();
                System.out.print("Introduzca el nombre: ");
                nom = In.readLine();
                System.out.print("Introduzca el modelo: ");
                mod = In.readLine();
                System.out.print("Introduzca el talle: ");
                talle = In.readInt();
                System.out.print("Introduzca el color: ");
                col = In.readLine();
                System.out.print("Introduzca la cantidad: ");
                can = In.readInt();
                System.out.print("Introduzca el precio: ");
                pre = In.readFloat();

                items[i].SetRopa(cod, nom, can, pre, talle, mod, col);
        }
   }

   void VerLista () {
        System.out.println(titulo);
        System.out.println("Codigo Nombre Modelo Talle Color Cant Precio
                           Total");
        for (int i=0; i<elem; i++)
        {
                System.out.println(items[i].GetCodigo()+ " " +
                items[i].GetNombre()+ " " +
                items[i].GetModelo()+ " " +
                items[i].GetTalle()+ " " +
                items[i].GetColor()+ " " +
                items[i].GetCantidad()+ " " +
                items[i].GetPrecio()+ " " +
                items[i].TotalArt() );
        }
        System.out.println("-----------------------------------------" );
        System.out.println("Cantidad Total : " + TotalCant() );
        System.out.println("Precio Total   : " + TotalPrecio() );
        System.out.println("Total General  : " + TotalGeneral() );
   }
};


//archivo: ListaApp.java

import java.io.*;
import utn.frc.io.In;

public class ListaApp {
   public static void main(String[] args) {
       int n;
        String tit;
        System.out.print("Ingrese el nombre del listado :");
        tit = In.readLine();
```

```
            System.out.print("Ingrese el tamaño de la lista: ");
            n = In.readInt();

            LStock lista = new LStock(n,tit);

            lista.CargarLista ();
            lista.VerLista ();
        }
    }
```

11. En un sistema para la fabricación y comercialización de juguetes encontraron lass siguientes entidades durante la etapa de análisis:

Artículo: entidad generica que posee como atributos: material y precio, como comportamiento retornar datos, retornar precio y getserie() el cual no tiene implementación.

Auto: Entidad que extiende de Articulo, agrega cant de puertas y numero de serie de auto (generado en forma automática por el sistema), la cantidad de puertas aumenta el precio en 1.5$ por cada puerta extra (siendo las extras las que superan a dos puertas).

Moto: Entidad que extiende de Artículo, agrega cant de ruedas (2,3 o 4) y numero de serie de moto (generado en forma automática por el sistema, el precio se inrementa según la siguiente tabla:

2 ruedas 1.50$

3 ruedas 3.20$

4 ruedas 5.60$

Implementar en las dos clases anteriores getserie() que devuelva el numero de serie del auto o moto respectivamente. Desarrollar constructor con parámetros y copia.

Declarar en el main() un vector de 50 referencias a la clase Articulo, en cada posición del vector puede haber Autos o Motos elegidos por el usuario. Implementar desde todos sus componentes los métodos desarrollados.

```
//archivo: Articulo.java


public abstract class Articulo {
   protected String material;
   protected float precio;

   public Articulo ( String mat, float pre) {
       material = mat;
       precio = pre;
   }

   public Articulo (Articulo a) {             //constructor copia
       material = a.material;
       precio = a.precio;
   }

   public String RetornarDatos () {     return (material + " " +  precio); }

   public abstract float RetornarPrecio ();

   public abstract int getSerie();

}//fin clase Articulo

//archivo: Auto.java

public class Auto extends Articulo {
   protected int canPuertas;
   protected static int numSerie = 0;

   public Auto ( String mat, float pre, int can) {
       super(mat, pre);
```

```
        canPuertas = can;
        numSerie++;
    }

    public Auto ( Auto a ) {
        super(a.material, a.precio);
        canPuertas = a.canPuertas;
        numSerie = a.numSerie;
    }

    public String retornarDatos () {
        String datos = material+" "+canPuertas+" "+numSerie+" "+RetornarPrecio();
        return datos;
    }

    public float RetornarPrecio ()
    {
        float nprecio = precio;
        if (canPuertas > 2)
               nprecio += (1.50 * canPuertas);
        return nprecio;
    }

    public int getSerie() { return numSerie; }
}//fin clase Auto

//archivo: Moto.java

public class Moto extends Articulo {
    protected int canRuedas;
    protected static int numSerie = 0;

    public Moto ( String mat, float pre, int can)
    {
        super(mat, pre);
        canRuedas = can;
        numSerie++;
    }

    public Moto ( Moto a )
    {
        super(a.material, a.precio);
        canRuedas = a.canRuedas;
        numSerie = a.numSerie;
    }

    public String retornarDatos ()
    {
        String datos = material+" "+canRuedas+" "+numSerie+" "+RetornarPrecio();
        return datos;
    }

    public float RetornarPrecio ()
    {
        float nprecio = precio;
        if (canRuedas == 2)
               nprecio += 1.50;
        if (canRuedas == 3)
               nprecio += 3.20;
        if (canRuedas == 4)
               nprecio += 5.60;
        return nprecio;
    }

    public int getSerie() { return numSerie; }
}//fin clase Moto
```

```
//archivo: EjemploApp.java

import java.io.*;
import utn.frc.io.In;

public class EjemploApp {
   public static void main(String[] args) {
       int i;
        Articulo a[] = new Articulo[2];

        for (i=0 ; i<2 ; i++) {
                int op, c;
                float p;
                String m;
                System.out.print("¿Fabrica un Auto (1) o Moto (2)?: ");
                op = In.readInt();
                if (op == 1)
                {
                    System.out.print("Introduzca el material: ");
                        m = In.readLine();
                        System.out.print("Introduzca el precio: ");
                        p = In.readFloat();
                        System.out.print("Introduzca el cant. de puertas: ");
                        c = In.readInt();
                    a[i] = new Auto(m, p, c);
             }
                else
             {
                       System.out.print("Introduzca el material: ");
                        m = In.readLine();
                        System.out.print("Introduzca el precio: ");
                        p = In.readFloat();
                        System.out.print("Introduzca el cant. de ruedas: ");
                        c = In.readInt();
                       a[i] = new Moto(m, p, c);
             }
        }

        for (i=0 ; i<2 ; i++) {
                System.out.println("Los datos completos son: "+
                a[i].RetornarDatos());
                System.out.println("Precio : "+ a[i].RetornarPrecio());
                System.out.println("Numero de serie: "+ a[i].getSerie ());
       }
   }
}
```

EJERCICIOS PARA RESOLVER

1. Desarrolle la clase **CajaDeAhorros** que permita manejar una cuenta caja de ahorros con los siguientes atributos:

- nombre (del cliente)
- tipo ('C' = común 'E'= especial)
- saldo
- cantidad (indica la cantidad de extracciones **realizadas**, sabiendo que si el tipo es común puede realizar 3 extracciones pero si es especial puede realizar 5 extracciones)

Desarrollar los métodos:

- **cargar**

- **mostrar**
- **depositar**(cuyo monto a depositar será pasado por parámetro – actualizando el saldo)
- **extraer** (cuyo monto a extraer será pasado por parámetro – verificando si puede o no extraer – en caso afirmativo actualizar el saldo)

Desarrollar un menú de opciones en el programa principal para la invocación de los distintos métodos.

2. Cree una clase llamada **Hora** que tenga miembros datos separados de tipo int para horas, minutos y segundos. Un *constructor* inicializará cada atributo a 0, y otro lo inicializará a valores parametrizados. Una función miembro deberá visualizar la hora en formato 11:20:30. Otra función miembro sumará dos objetos de tipo hora pasados como argumentos. En una función principal main() cree dos objetos inicializados y uno que no esté inicializado. Sume los dos valores inicializados y deje el resultado en el objeto no inicializado. Por último visualice el valor resultante.

3. Desarrollar la clase Teclado cuyos atributos son los siguientes:

- marca
- canTeclas
- tipo (1-español, 2-latinoamericano, 3-estados unidos)

Además, desarrollar el/los constructor/es correspondiente/s a la clase, que permita/n inicializar los atributos con los datos que se pasen como parámetros. De no pasarse ningún valor como parámetro los objetos deberán inicializarse con los siguientes valores

marca: "BTC"

canTeclas: 104

tipo: 1

- Desarrollar el método modTipo, que modifique el tipo de teclado por uno distinto al que ya tiene.
- Desarrollar el método mostrar, que muestre todos los atributos.
- Desarrollar el método mostrar(int tip), que muestre todos los atributos siempre y cuando coincida el tipo pasado por parámetro con el tipo de teclado del objeto.

Luego implementar un main() donde se creen objetos de la clase teclado

4. Cree una clase llamada **Empleado** que contenga como miembro dato el nombre y el legajo del empleado, y como funciones miembro leerDatos()que lea los datos desde teclado y verDatos() que los visualice en pantalla. Escriba un programa que utilice la clase, creando un array de tipo Empleado y luego llenándolo con datos correspondientes a **N** empleados. Una vez rellenado el array, visualice los datos de todos los empleados.

5. Realizar un programa que calcule la edad (cantidad de años) de una persona, de manera que ésta ingrese su fecha de nacimiento (Día, Mes, Año) y el programa le devuelva su edad.

6. Crear una clase (Emplado) que contenga el legajo y los 12 sueldos de un empleado. Poner el vector de sueldos en cero mediante el constructor, crear los métodos cargar, mostrar datos, toString que retorne todos los datos del empleado e implementar ésta clase en el main().

7. Los números racionales o fraccionarios son muy usados en la vida real, generalmente los utilizamos al medir cantidades, distancias, tiempo, pesos, entre otras cosas. Java no trae incorporado un tipo de datos que represente a un fraccionario como un numerador / denominador, por lo que les proponemos que ustedes definan una clase racional para poder representar a éste tipo de números.

El objetivo es implementar una calculadora de racionales muy primitiva con las operaciones

básicas: suma, resta, multiplicación, división, comparación de fracciones.

Para llegar al objetivo se deben cumplir las siguientes actividades:

- Definir los atributos de un número fraccionario
- Implementar la clase fraccionario, sobrecargando los operadores que permitan realizar las opresiones solicitadas
- Definir un constructor con parámetros y uno por defecto
- Definir un método Mostrar que imprima en pantalla el número en forma de fracción, por ejemplo " 1/2"
- Una clase Main: Desarrollar un menú que permita:
  1. Ingresar dos números fraccionarios
  2. Sumar los dos números.
  3. Restar los dos números.
  4. Multiplicar los dos números.
  5. Dividir los dos números.
  6. Verificar si los dos números son iguales.
  7. Salir del programa

Nota: se deben mostrar por pantalla todos los resultados de las operaciones.

8. En una agencia publicitaria se ha organizado el tratamiento de los datos sobre publicidades contratadas de la siguiente manera:

Clase publicidad:

- Atributos: Nombre, Medio, costo

Clase empresa:

- Atributos: Nombre, Rubro, Publicidad: Vector de 10 publicidades.

Se deben definir e implementar los siguientes métodos:

- ✓ Constructores: la clase empresa debe tener un constructor con parámetros y uno por defecto, la clase publicidad solamente un constructor con parámetros
- ✓ Destructores: donde sea necesario.
- ✓ Mostrar datos para ambas clases (Una empresa también debe mostrar los datos de sus publicidades asociadas)
- ✓ Crear nueva publicidad en la clase empresa
- ✓ Costo total para la clase empresa que devuelva el costo de todas las publicidades de una determinada empresa
- ✓ Buscar una publicidad de la empresa por el nombre de la misma y devolver 1 si la encuentra y 0 en el caso contrario.
- ✓ Buscar una publicidad de la empresa por el medio por la que se realizó y devolver su nombre.

Nota: Definir e implementar más métodos si lo considera necesario.

En la clase main se debe instanciar al menos una empresa contratante y llamar a los métodos definidos anteriormente. También puede (opcionalmente) definir un menú de opciones para utilizar los mismos.

9. Se necesita implementar a través de la programación orientada a objetos el manejo de los artículos de una fábrica de gaseosas. La fabricación de los mismos se realiza en cinco pasos predeterminados cada uno de los cuales consume un tiempo en particular. Los datos de cada articulo son:

- ✓ Codigo_articulo
- ✓ Descripción
- ✓ Fecha de elaboración
- ✓ Tiempos utilizados (Para representar los mismos se puede utilizar una matriz de cinco filas y tres columnas donde cada fila represente un paso y las columnas representen: tiempo estimado / tiempo real utilizado / tiempo de retraso=t.estimado-t.real)

La fábrica desea poder realizar las siguientes operaciones (main)

- ✓ Cargar el vector de artículos, el usuario debe poder ingresar el tamaño del mismo.
- ✓ Mostrar los datos actuales de un articulo (ingresando su código) y permitir la modificación de la descripción y de la fecha de elaboración
- ✓ Dar el promedio general del tiempo real en que tarda en ser fabricado para cada uno de los artículos.
- ✓ Listar código, descripción y fecha de aquellos artículos hayan vencido (La fecha de vencimiento de un articulo corresponde a 2 años posteriores a su fecha de elaboración)
- ✓ Listar las descripciones de aquellos artículos que se hayan retrasado en algún paso de su elaboración

Nota: para la fecha de elaboración utilizar la clase fecha vista en clases de práctico.

10. El gerente de un centro comercial desea contar con un programa que le permita el manejo de los locales que va alquilando. Cada comercio tiene como datos: un número de local asignado por el propio gerente, un nombre y un monto de facturación mensual. Debe desarrollarse un menú que permita:

1-Incluir un nuevo comercio al darse un alquiler (ALTA)

2-Excluir un comercio ingresando su número como dato (BAJA)

3-Listar por pantalla los datos de todos los locales alquilados

4-Mostrar los datos de un negocio ingresando y buscando según su nombre

5-Mostrar los datos de aquellos negocios cuya facturación sea mayor a cierto monto ingresado como parámetro

11. Se necesita implementar a través de la programación orientada a objetos, el manejo de la lista de alumnos de una unidad académica de la UTN. Los datos de cada alumno a tener en cuenta son:

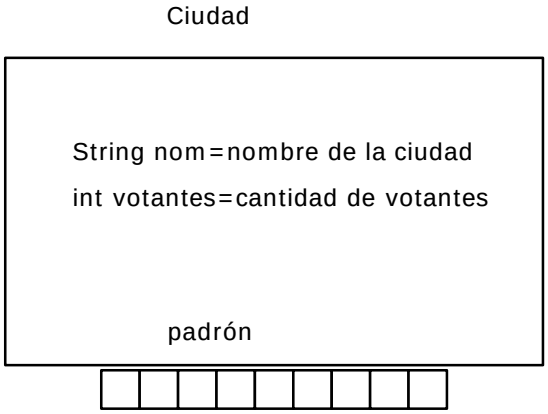
- Legajo
- Nombre y apellido
- Fecha de nacimiento
- Calificación obtenida en los tres parciales y el examen final de cada materia de su carrera (se trata de 42 materias en total) Se recomienda usar una matriz de 42 materias por 4 columnas de parciales y final) o un objeto que permita manejar esta información.

Presentar un menú con las siguientes opciones:

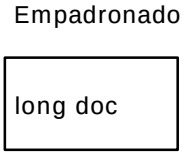
- Insertar un alumno nuevo
- Dar de baja un alumno
- Mostrar los datos actuales (de un alumno buscado por legajo) y luego modificarlos.
- Dar el promedio general (exámenes finales) de cierta materia.
- Listar legajo, nombre, apellido y edad de aquellos alumnos con todas sus materias rendidas y aprobadas.

12. La organización de los votantes de las diferentes ciudades para una elección, se efectúa mediante

el uso de la siguiente definición de clases:



Donde "padrón" es un vector en el cual cada elemento del vector es a su vez un objeto de tipo "empadronado" de la forma:



A través de un main() y de la definición de los métodos pertinentes para cada clase se deberán completar los datos de un objeto "ciudad" para luego poder mostrar los resultados.

13. El dueño de una guardería de niños necesita realizar un sistema que lleve un registro de todos los alumnos inscriptos. Los datos que le interesa manejar de cada niño incluyen:

- legajo
- nombre del alumno
- nombre del padre
- domicilio
- teléfono
- fecha de nacimiento

Se debe considerar que periódicamente se inscriben nuevos alumnos en la guardería, y que los legajos de los alumnos no son otorgados por el cargador de datos, sino que deben ser asignados automáticamente por el sistema al darse el alta.

El programa realizado debe utilizar una clase Alumno que permita manejar los datos de cada escolar individual y otra clase Curso que maneje una lista con todos los alumnos que realizan su instrucción en la institución. Se deben definir, además, en la clase que corresponda los métodos necesarios para realizar las siguientes tareas:

- constructores donde sea necesario.
- poder eliminar un alumno, conociendo únicamente su número de legajo o su nombre.
- visualizar los datos de un alumno, conociendo su número de legajo o el nombre del alumno.
- crear un alumno e insertarlo en la lista.

- devolver la cantidad de alumnos que han cursado en la guardería (último número de legajo).
- definir constructor copia para la clase Curso.
- Escribir un main() que implemente el funcionamiento definido en las clases, con un pequeño menú.

14. Para organizar mejor los datos de sus alumnos, el mismo dueño de la misma guardería del ejercicio anterior, ha determinado dividirlos en distintos cursos de acuerdo a la edad de cada pequeño. La guardería recibe niños de 0 a 9 años, por lo que se crearán 10 grupos distintos.

Para ello se deberá definir una clase Guardería que maneje como información el nombre, la dirección, el teléfono y el nombre de su dueño. Además de esto, deberá mantener un vector de al menos 10 referencias a listas de alumnos del ejercicio anterior (clase Curso), clasificada por la edad de sus miembros.

Asimismo, se deben definir constructor y métodos para la clase Guardería, y los métodos necesarios para:

- insertar un alumno en la lista que le corresponda de acuerdo a su edad.
- eliminar un alumno de su lista, teniendo como información el legajo del alumno.
- contar la cantidad de alumnos de una determinada edad.
- modificar los datos de un alumno, ubicándolo por su legajo.

15. Se necesita implementar a través de la programación orientada a objetos el manejo de una lista de artículos de una fábrica de gaseosas. La fabricación de los mismos se realiza en cinco pasos predeterminados cada uno de los cuales consume un tiempo en particular. Los datos de cada articulo son:

- ✓ Codigo_articulo
- ✓ Descripción
- ✓ Fecha de elaboración
- ✓ Tiempos utilizados (Para representar los mismos se puede utilizar una matriz de cinco filas y tres columnas donde cada fila represente un paso y las columnas representen: tiempo estimado / tiempo real utilizado / tiempo de retraso=t.estimado-t.real)

Para manipular estos datos se debe realizar en el punto de entrada un menú de opciones

- ✓ Insertar un articulo nuevo (Donde se debe tener en cuenta que el código de articulo no es otorgado por el cargador de datos sino que el programa lo genera automáticamente al producirse un alta. El numero generado es el correlativo superior del ultimo ingresado)
- ✓ Dar de baja un articulo
- ✓ Mostrar los datos actuales de un articulo (ingresando su código) y permitir la modificación de los mismos
- ✓ Dar el tiempo total y cada uno de los tiempos parciales en que se fabrico un determinado articulo
- ✓ Dar el promedio general (de todos los artículos de la lista) del tiempo real en que tarda un articulo en ser fabricado.
- ✓ Listar código, descripción y fecha de aquellos artículos hayan vencido (La fecha de vencimiento de un articulo corresponde a 2 años posteriores a su fecha de elaboración)
- ✓ Listar las descripciones de aquellos artículos que se hayan retrasado en algún paso de su elaboración

16. El procesamiento de los resultados de una elección municipal de la cual participaron solo dos partidos (A y B), se efectúa mediante el uso de la siguiente definición de clases:

NRO: número de seccional

CANFEM: total padrón femenino

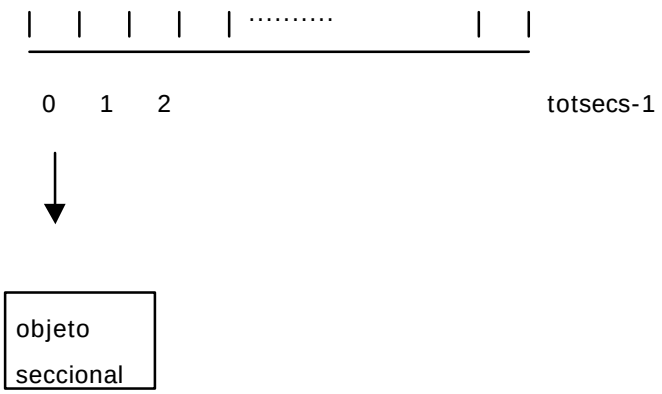
Seccional CANMASC: total padrón masculino

Static SECS_PROC: dato propio de la clase que representa el total de seccionales que se llevan procesadas.

primero: referencia a cod_partido/tipo_padrón/votos/prox (otra clase)

Elección S[totsecs]: donde, si definimos que totsecs=17, establecemos ese número de referencias a objetos de la clase seccional

Podríamos presentarlo



A través de un main() y de la definición de los métodos pertinentes, se deberán completar los datos de un objeto elección para luego poder mostrar sus resultados.

17. Crear una clase notas en la cual cada objeto contiene el nombre y la nota de un alumno en una determinada materia. Crear los métodos necesarios para cargar y visualizar los datos. Crear además un método estático que calcule el promedio de las notas de todos los alumnos ingresados. Implementar todo esto con un vector de referencias a objetos.

18. En una agencia publicitaria se ha organizado el tratamiento de los datos sobre publicidades contratadas de la siguiente manera:

```
Class Publicidad
{ String nombre, nombre_medio;
  float costo;
    ........
};

class Empresa_contratante
{ String nombre_empresa;
  String rubro;
  Publicidad p[]= new Publicidad[10];
    . . . ..
```

};

Ésta agencia ha determinado que puede generar solo 10 publicidades por cada empresa, existiendo la posibilidad de que una empresa determinada tenga menos de 10 publicidades; por lo tanto las publicidades de una empresa se van creando a medida que se van necesitando.

Se deben definir e implementar los siguientes métodos:

- ✓ Constructores
- ✓ Mostrar datos para ambas clases (Una empresa también debe mostrar los datos de sus publicidades asociadas)
- ✓ Crear nueva publicidad en la clase empresa
- ✓ Costo total para la clase empresa que devuelva el costo de todas las publicidades de una determinada empresa

Nota: Definir e implementar más métodos si lo considera necesario.

En el punto de entrada se debe instanciar al menos una empresa contratante y llamar a los métodos definidos anteriormente. También puede (opcionalmente) definir un menú de opciones para utilizar los mismos.

19. Crear una clase persona que contenga su DNI, nombre y edad. Para manejar éstos datos tendrá los siguientes métodos:

- Constructor con parámetros
- Constructor copia
- Cargar()
- Mostrar()
- toString()

Implementar la clase con una lista de objetos.

20. Desarrollar la clase **ListaArticulos** para manejar una lista de artículos, donde cada artículo tiene la siguiente información

- codigo (int)
- descripción (cadena de caracteres)
- precioUnit (float)

Como único atributo la clase tendrá una referencia a una lista (lineal simplemente vinculada).

Métodos:

- **crearNodo**: creará un nodo con la información correspondiente a un artículo.
- **mostrarListado**: mostrara por pantalla todo el listado de artículos.
- **buscar(int cod)**: buscará un código de articulo en la lista, si está informará todos sus datos; caso contrario imprimirá un mensaje de error.
- **int cantidad()**: debe devolver la cantidad de artículos que hay en la lista.
- **Constructor**: que inicialice la lista

Implementar un main() con menú de opciones para la invocación de los distintos métodos para el manejo de una lista de artículos (o sea, crear un solo objeto de la clase ListaArticulos)

EJERCICIOS CON HERENCIA

21. Para administrar las compras realizadas con tarjetas en un negocio, se debe desarrollar un programa orientado a objetos que pueda manejar mediante clases las siguientes entidades descubiertas durante la etapa de análisis del sistema:

- **Tarjeta**: es una tarjeta genérica que tiene las siguientes propiedades:
  - número
  - titular
  - cuotas: cantidad máxima de cuotas que permiten los planes de pago de esta tarjeta

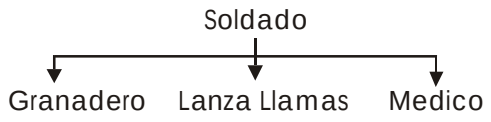
  y debe poseer las operaciones que se indican:
  - constructor con parámetros que inicialice todos los atributos
  - void mostrar() que muestra todos los datos de la tarjeta
  - float calcCuota(float precio, int pagos) que retorne el valor de la primera cuota
- **TDebito**: tarjeta de débito (Banelco) que puede ser utilizada como dinero electrónico para realizar compras a precio de contado. Las características de este tipo de tarjetas incluyen todas las de la entidad anterior, más las siguientes propias:
  - número de cuenta: número de la cuenta bancaria asociada
  - saldo: saldo en la cuenta asociada a la tarjeta

  y debe poseer las siguientes operaciones:
  - constructor con parámetros que inicialice todos los atributos
  - void mostrar()que muestra todos los datos de la tarjeta de débito
  - float calcCuota(float precio, int pagos) que retorne el valor de la primera cuota, considerando que dicho monto no puede exceder el saldo que posee, que la cantidad de cuotas no puede ser mayor al límite que la tarjeta tiene fijado, y que recibe un descuento del 5% del IVA (paga sólo 16% de IVA).
  - void suma() para que sume los saldos de dos tarjetas de débito.
- **TCredito**: tarjeta de crédito que puede ser utilizada para realizar compras financiadas. Las características de este tipo de tarjetas incluyen todas las de la clase Tarjeta, más las siguientes propias:
  - marca: nombre comercial de la tarjeta de crédito (ej.: VISA, Mastercard, etc.)
  - recargo: porcentaje de sobreprecio cobrado por el comercio a las compras con la tarjeta de crédito

  y debe poseer las siguientes operaciones:
  - constructor con parámetros que inicialice todos los atributos
  - void mostrar()que muestra todos los datos de la tarjeta de crédito
  - float calcCuota(float precio, int pagos) que retorne el valor de la primera cuota, considerando que la cantidad de cuotas no puede ser mayor al límite que la tarjeta tiene fijado, y que recibe el recargo indicado sobre el precio total del bien o servicio comprado.

Hacer un pequeño main que represente el uso que una persona realiza de sus 3 tarjetas de crédito y 1 de débito, manejándolas con un vector de referencias a la clase base Tarjeta.

22. La empresa Westwood Inc. (creadora de juegos tipo Red Alert) nos encargo la creación de un nuevo juego, el cual debe respetar el estilo de sus antecesores. El libreto exige la creación de los siguientes personajes con la jerarquía de clases propuesta



Donde :

Soldado posee como dato miembro *Color, Cantidad de vida* (Vida), *Velocidad de desplazamiento, Cantidad de balas* (hasta 40). Y las funciones miembro son *Recargar(int cant)* (esta función se ejecuta cuando el soldado encuentra balas), *Disparar(int cant*) (cantidad de disparos), *Alcanzado(int valor)* (esta función es utilizado cuando el soldado es herido y recibe como parámetro la cantidad de vida que pierde, retorna 1 si se muere -vida <=0-), curar*()* (este método es utilizado cuando el medico asiste al soldado), constructor con parámetros por defecto.

Granadero **que agrega a las características de soldado *Cantidad de bombas* (hasta 20), *armadura* (una segunda protección). Y reutiliza las funciones *alcanzado(int valor)* (que distribuye en partes iguales la disminución de vida y armadura*), Recargar(int cant*) (teniendo en cuanta las restricciones) y su respectivo constructor con parámetros por defecto.**

Lanza Llamas que agrega a las características de soldado *nivel de combustible* (que puede lanzar, este dato es un porcentaje), *armadura* W.2 Can0.127612Tc 2.4733 Tw () (cantidad 0 Tw Tw6855s

- **status**, que muestra por pantalla los valores de sus variables internas.

Dentro de este marco común son construidos los satélites comerciales y militares. Estos poseen las siguientes características particulares.

- **Satélite comercial**:
  - el **alcance de la antena** en este tipo de satélites es un valor definido para cada satélite cuando es armado. Para saber si algún objeto a su alcance se debe calcular por Pitágoras la distancia de este objeto con respecto a las coordenadas del satélite.
  - además puede **sintonizar un canal**, para lo que recibe un entero y una frecuencia y devuelve el producto de ambos luego de ajustar sus componentes electrónicos.
- **Satélite militar**:
  - el **radio de alcance del láser** utilizado como guía de misiles.
  - además puede **disparar láser**, que recibe un entero indicando la potencia del láser y retorna 1 si el último blanco localizado puede ser alcanzado y 0 en caso contrario. La potencia reduce el alcance en un 10% por cada 100 unidades de potencia, arriba de los 1000 (o sea que una potencia de 1300 reduce el alcance del láser en un 30%).

Se deben implementar las clases necesarias para manejar estos satélites, con constructores con parámetros. Escribir un pequeño main() considerando que la implementación del sistema se lleva a cabo para un máximo de 50 satélites, los cuales se dividen en partes iguales en comerciales y militares.

24. Recientemente, se ha instalado en nuestra ciudad una cadena multinacional de restaurantes la cual desea implementar un sistema para su manejo interno. Se cuenta con los siguientes datos resultantes del relevamiento:

- ✓ Existen cuatro tipos de platos: Entradas, Plato Principal y Postre además un Menú del Día que a su vez esta compuesto por: Entrada, Plato Principal y Postre.
- ✓ Cada Plato tiene un nombre, precio, código de receta y cantidad vendida
- ✓ Las preguntas mas frecuentes y que por lógica necesitan mas rápida respuesta son: Cual es el menú del día y que precio tiene. Por ende se deben implementar los métodos mostrar y calcular el precio, además de los constructores con parámetros.
- ✓ El menú del día se carga día por día según la disponibilidad de materia prima conseguida en el mercado. Lo mismo sucede con los platos individuales.
- ✓ La capacidad máxima de producción es de 100 platos por día (ya sea simples o menú)
- ✓ Al final del día se debe realizar un arqueo de caja como en todas las otras sucursales calculando el total vendido y realizando los descuentos correspondientes a la comisión del mozo (6 %). El arque es enviado vía módem a la central mundial de la cadena, con un encabezado con los sig. Datos: Código de País, Sucursal, Dirección (del restaurante)
- ✓ Además se debe poder consultar un listado de las cantidades consumidas por cada tipo de plato en cualquier momento del día.

Su tarea es implementar la clase base: Plato y las clases Postre, Menú y Restaurante, ejemplificando su uso en un pequeño método main.

25. Para poder luchar contra el tráfico ilegal, la SIDE nos solicita un programa que pueda simular el comportamiento de traficantes (de armas y de estupefacientes) y carteles.

Hecho el diseño se ha detectado que se deberán implementar las siguientes clases:

Clase madre:

Traficante: con los siguientes atributos: alias, país de operación.

Clases hijas:

Traficante de Armas: alias, país de operación, vector de tipos de armas vendidas (cadenas), vector con la cantidad estimada en stock de cada tipo de arma, vector con los precios unitarios de cada arma (en los 3 últimos vectores las posiciones correspondientes serán del nombre, cantidad y precio de una misma arma)

Traficante de Drogas: alias, país de operación, droga que vende, cantidad de kilogramos almacenados, precio por gramo

Para estas clases se necesitan los siguientes métodos:

- Constructores con parámetros.
- mostrar() que muestra los datos de cada clase
- float vender(int): que disminuye el stock y devuelve el valor total de la venta realizada, según sea:
  - Traficante de Armas: vende 1 unidad del arma que se encuentra en la posición recibida por parámetro. Considerar que si no posee ese tipo de armas, no puede cobrar nada.
  - Traficante de Drogas: vende la cantidad de gramos recibida por parámetro de la droga que distribuye. Considerar que si no tiene droga en stock no puede cobrar nada, y si no le alcanza para cubrir el pedido vende todo lo que le queda.

Además se necesita la clase **Cartel** que tenga los siguientes atributos: nombre y un vector de 50 traficantes de armas y/o de drogas; y los siguientes métodos:

- cargar(): que carga por teclado un traficante de armas o de drogas en el vector.
- mostrar(): que muestra los datos de los objetos cargados.
- vender(int pos, int val): que realiza una venta para el traficante que esté en la posición **pos**, pasándole el valor recibido por parámetro a dicho traficante.

Se le pide desarrollar las clases y un método main() en Java

26. Para administrar los empleados de una empresa, se debe desarrollar un programa orientado a objetos que pueda manejar mediante clases las siguientes entidades descubiertas durante la etapa de análisis del sistema:

- **Empleado**: es un empleado genérico, que posee las propiedades enumeradas a continuación:
  - legajo
  - nombre
  - sección (1, 2 ,3 ó 4)

  y debe permitir realizar las operaciones que se indican:
  - void cargarDatos() que lee valores desde teclado, asignándolos a los atributos del empleado
  - void mostrarDatos() que muestra los valores de los atributos del empleado
  - float calcSueldo() que retorne el sueldo cobrado por el empleado
- **Mensualizado**: empleado que cobra un salario fijo por mes, sin importar la cantidad de días trabajados. Las características de este tipo de empleados incluyen todas las de la entidad anterior, más las siguientes propias:
  - antigüedad: cantidad de años que lleva trabajando en la empresa
  - bruto: monto bruto de sueldo que cobra por mes
  - descuento: porcentaje de descuento que se le aplica sobre el bruto al sueldo del empleado

  y debe permitir realizar las siguientes operaciones:

- void cargarDatos() que lee valores desde teclado, asignándolos a los atributos del empleado
- void mostrarDatos() que muestra los valores de todos los atributos del empleado mensualizado
- float calcSueldo() que retorne el sueldo cobrado por el empleado

- **Jornalizado**: empleado que cobra un salario dependiendo de la cantidad de días trabajados. Las características de este tipo de empleados incluyen todas las de la clase Empleado, más las siguientes propias:
  - duración_contrato: plazo en días del contrato del empleado
  - días_trabajados: cantidad de días que trabajó el empleado en el último mes
  - jornal: pesos que cobra por día trabajado

  y debe permitir realizar las siguientes operaciones:
  - void cargarDatos() que lee valores desde teclado, asignándolos a los atributos del empleado
  - void mostrarDatos() que muestra los valores de todos los atributos del empleado jornalizado
  - float calcSueldo() que retorne el sueldo cobrado por el empleado

Realizar en todos los casos :
- Constructor sin parámetros
- Constructor con parámetros
- Constructor copia

Se sabe que la empresa dispone de distintas fábricas distribuidas por el país, y en algunos casos no se sabe con exactitud de cuántos empleados dispone la fábrica y en otros ese tamaño es perfectamente conocido, por lo cual se diseñó la siguiente estructura de clases:

- **Fábrica** : es una fábrica genérica, no se esperan instancias de esta clase, debe permitir realizar las operaciones que se indican:
  - void mostrarDatos() que muestra los valores de los atributos de lo empleados
  - float calcSueldoPromedio() que retorne el sueldo promedio cobrado por los empleados
  - int cuantos(Empleado & X) que retorna la cantidad de empleados que trabajan en la misma sección del empleado X
  - void agregarUno() función que agrega un empleado en la lista.
- **Lista_Fábrica**: Clase que implementa una lista simplemente enlazada que posee una cantidad no conocida de empleados, y define todos los métodos antes mencionados.
- **Arreglo_Fábrica**: Clase que implementa un arreglo (arreglo de punteros a objetos empleado) que posee una cantidad conocida de empleados, ésta no supera los 200, y define todos los métodos antes mencionados.

Realizar en todos los casos:
- Constructor sin parámetros
- Constructor con parámetros

Implementar para el manejo de toda la empresa la clase:

Empresa: Clase que maneja un arreglo de punteros a 10 fábricas además de:
- Razón social
- Nombre
- Dirección de casa central

y debe permitir realizar operaciones tales como
- void mostrarDatos() función que muestra todos los datos de la empresa, incluyendo todos los de las fábricas
- void cargarDatos() función que genera una nueva fábrica en el arreglo.
- void cargarDatos(int p) función que genera un nuevo empleado en la fábrica que se encuentra en la posición p dentro del vector.
- void cargarDatos(int c, int p) función que genera c nuevos empleado en la fábrica que se

encuentra en la posición p dentro del vector.

- float sueldoPromedio() función que retorna el sueldo promedio cobrado por todos los empleados de la empresa.
- int seccion(Empleado X) función que retorna la cantidad de empleados en la fábrica que trabajan en la sección X.

Ejemplificar todo lo anterior en un pequeño main().

27. Nos han pedido que desarrollemos un pequeño programa para administrar las fichas de los distintos objetos expuestos en el museo Rocksen. Estos objetos pueden ser de 2 tipos: históricos (piedras, fósiles, plantas, momias, etc.) u obras de arte (esculturas, pinturas, etc.). Durante la etapa de análisis del sistema se descubrieron las siguientes entidades:

- **Objeto**: es cualquier elemento expuesto, que posee las siguientes propiedades:
  - código
  - nombre
  - año: en que fue realizada la obra o estimación de formación/creación de elementos naturales

  y debe poseer las operaciones que se indican:
  - constructor con parámetros que inicialice todos los atributos
  - String getNombre() que retorna el nombre del objeto
  - String getDatos() que retorna todos los datos del objeto en exposición
- **Histórico**: elemento antiguo expuesto. Las características de este tipo de objetos incluyen todas las de la entidad anterior, más las siguientes propias:
  - zona: cadena de caracteres que indica la zona geográfica en la que fue encontrado
  - origen: cadena de caracteres que describe el origen del objeto

  y debe poseer las siguientes operaciones:
  - constructor con parámetros que inicialice todos los atributos
  - String getNombre() que retorna el nombre del objeto, seguido por la zona, ej: león (África).
  - String getDatos() que retorna todos los datos del objeto en exposición
  - void mostrarDatos() que muestra por pantalla todos los datos del objeto en exposición
- **Obra**: obra de arte expuesta. Las características de este tipo de objetos incluyen todas las de la clase Objeto, más las siguientes propias:
  - autor: nombre del autor de la obra
  - país: país de origen de la obra

  y debe poseer las siguientes operaciones:
  - constructor con parámetros que inicialice todos los atributos
  - String getNombre() retorna el nombre de la obra, seguido por su autor, ej: Girasoles (Van Gogh).
  - String getDatos() que retorna todos los datos de la obra de arte
  - void mostrarDatos() que muestra por pantalla todos los datos de la obra de arte
- **Sala**: sala de exposiciones que puede tener hasta 100 objetos de cualquier tipo en exposición y contiene los siguientes atributos:
  - número
  - ala: ala del museo donde se ubica la sala (Norte o Sur)
  - objetos: vector de 100 punteros a Objeto para los elementos en exposición

  y debe poseer las siguientes operaciones:
  - constructor con parámetros que inicialice los atributos (sólo inicializa atributos, no carga objetos)
  - int agregarObjeto(Objeto & o): agrega el objeto recibido al vector de objetos, si quedaba

alguna posición libre. Devuelve 1 si pudo agregarlo o 0 si no pudo

- void mostrarDatos(): muestra por pantalla los valores de los atributos propios y luego los datos completos de cada objeto cargado

Desarrollar un pequeño main() donde se ejemplifique el uso de las clases, creando una instancia de Sala y agregándole algunos objetos.

28. Una fábrica de dulces desea implementar un Sistema de Producción donde se manejen los siguientes datos:

- ✓ Golosina: código de golosina

  Nombre

  Precio

  Ingredientes

  Donde ingredientes es un vector de 15 punteros en el que cada uno podrá contener un nombre de ingrediente de dicha golosina. Ésta clase deberá contener métodos que permitan mostrar los datos, agregar un nuevo ingrediente a los existentes y modificar el precio.

- ✓ Caramelo: Clase que agrega a los datos de golosina:

  Sabor

  Color

  Y contiene un método para mostrar los datos

- ✓ Relleno: Clase que agrega a los datos de caramelo:

  Sabor del relleno

  Cantidad

  Y contiene un método para mostrar los datos

- ✓ Chocolate: Clase que agrega a los datos de golosina:

  Tipo (Solo puede ser blanco o negro)

  Fecha de vencimiento

  Y contiene un método para mostrar los datos y otro que devuelve la fecha de vencimiento si el precio del chocolate es menor a $10

Se pide:

-Definir las clases y métodos correspondientes.

-Definir los constructores que considere necesario.

-Definir en el main() un vector de 15 punteros donde los primeros 5 sean caramelos, otros 5 caramelos rellenos y el resto chocolates. Llamar a los métodos definidos desde éste vector.

29. En una agencia de turismo se ha organizado el tratamiento de los datos sobre viajes de la siguiente manera:

-VIAJE: posee int nro_viaje
String destino
int cant_de_dias
String hotel
float precio_por_dia

donde nro_viaje es un valor que se genera automáticamente al crear un nuevo viaje.

-VIAJE_EN_CURSO: Derivada de VIAJE, representa a aquellos viajes de la agencia que están

actualmente ejecutándose, por lo cual agregan a los datos de VIAJE:

String fecha_de_llegada

int cant_de_pasajeros

-VIAJE_EN_ESPERA: Derivada de VIAJE, sus objetos son viajes que saldrán en los próximos 60 días, por lo cual se agregan a los datos de VIAJE :

String fecha_de_salida

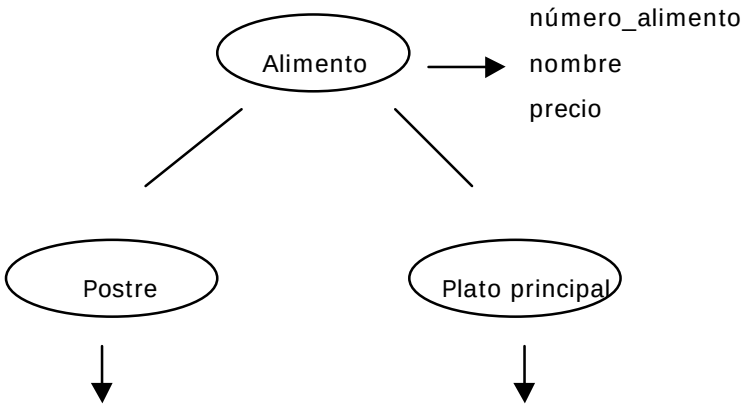
int cant_de_personas_confirmadas

int cupo_máximo

Se pide:

- Definir todas las clases teniendo en cuenta que la clase VIAJE nunca será instanciada.
- Definir constructores (con parámetros por defecto).
- Definir para la clase VIAJE el método "calcular_precio(int x)" que calcula el costo del viaje según la cantidad de días que se desee, donde x representa la cantidad de días (si no se ingresa valor para x se toma por defecto la cantidad total de días que dura el viaje).
- Definir para la clase VIAJE_EN_ESPERA la sobrecarga del operador <, que compara si la cantidad de espacios libres de un viaje es menor que la del otro. Espacios libre es la diferencia entre el cupo máximo y la cantidad de personas confirmadas.
- Definir para la clase VIAJE_EN_CURSO el constructor copia (teniendo en cuenta que los números de viaje no pueden ser iguales).
- Definir en el main() un vector de 10 referencias a la clase base, en el cual los primeros 5 contengan objetos de tipo VIAJE_EN_CURSO, y los 5 restantes de VIAJE_EN_ESPERA. Luego crear objetos de las clases derivadas e implementar los métodos anteriormente definidos.

30. Los datos de un restaurant de pastas de la ciudad se organizan de la siguiente manera:



País de procedencia	tiempo_de_demora
Tipo_postre (torta,helado,etc)	tipo_pasta (lasagna,ravioles,etc)
Extras(dulce de leche,crema,canela)	salsa (bolognesa,blanca,etc)

Nota: Al crearse un alimento, el número no se ingresa por teclado sino que el programa lo genera en forma automática de acuerdo a la cantidad de platos creados hasta el momento

Se pide:

- Definir las tres clases teniendo en cuenta que la clase alimento nunca se instanciará.
- Definir constructores (con argumentos).

- Definir para la clase alimento el método “modificar_precio( )” .
- Definir para las tres clases el método “mostrar( )”.
- Definir para la clase plato principal el operador > que compare entre dos platos cuál es el que tiene mayor tiempo de demora.
- Definir en el main() un vector de 10 referencias a la clase base, en el cual los primeros 5 contengan objetos de tipo postre, y los 5 restantes de plato principal; implementar los métodos definidos a partir del vector creado.

31. Se han organizado los datos de las agencias de turismo de una ciudad en clases de la siguiente manera:

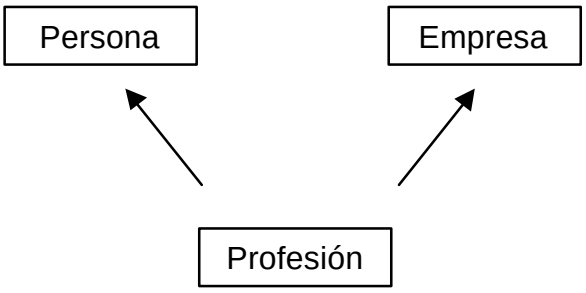
AGENCIA: posee String NBR, int MATRICULA.

AGENCIA_ACTIVA: derivada de AGENCIA, representa a las agencias que no solo están matriculadas sino que también operan en ésta temporada ppor lo cual posee como dato el puntero al primer nodo de una lista de destinos (nodo*primero). Cada nodo representa un destino trabajado por la agencia y guarda: String NOM (del destino), int CONTINENTE (0 Europa/ 1 Africa/ 2 Asia...) y el puntero al siguiente nodo.

Se pide

- Definir las clases e instancias de las mismas en el main().
- Definir e implementar constructores.
- Definir e implementar el método “AgregaDestino” con los parámetros necesarios para la clase AGENCIA_ACTIVA.
- Definir e implementar para la clase AGENCIA_ACTIVA, el método “DestinosxCont” que recibe como parámetro un código de continente y retorna la cantidad de lugares destino que se trabajan dentro del mismo.
- Declarar en el main() un vector de 5 agencias teniendo en cuenta que las correspondientes a las posiciones 0 y 3 están en actividad esta temporada.

31. Una serie de datos se organizan en las siguientes clases para representar la situación laboral y datos personales de un profesional:



Cada una de estas clases tiene los siguientes atributos:

Persona:	String	nombre	
	char	documento[10]	
	String	direccion	
	int	tipo_de_documento	(0 DNI/ 1 LE/ 2 LC/ 3 otro)
Profesional:	int	profesión	(0 abogado/ 1 ingeniero/ 2 contador/ 3 otra)

String puesto
float sueldo
long matricula

Empresa: String nombre
String direccion

Se pide:

- Definir las tres clases e instancias de las mismas en el main().
- Definir e implementar constructores:
- Definir e implementar los métodos Mostrar y Cargar para las tres clases.
- Definir e implementar en la clase Profesional un método que reciba como parámetros un código de profesión y un monto cualquiera, para imprimir los datos: nombre del profesional, matrícula, nombre de la empresa en la cual trabaja, puesto y sueldo; sólo si, teniendo esa profesión, cobra un sueldo mayor al monto pasado como parámetro. De no cumplir alguna de dichas condiciones, mostrar el mensaje necesario.
- Declarar un vector de 15 profesionales. Implementar con dicho vector todos los métodos definidos.
- Declarar un objeto de tipo persona. Llamar a partir de él a las funciones necesarias.

32. Una serie de datos se organizan en las siguientes clases:

Persona Conciento

Músico

Cada una de ellas dispone de:

- Persona: String NBR, long int DOC, int TIPO(0.DNI, 1.LE, 2.LC, 3.Otro)
- Músico: (además de los datos heredados desde persona): int TIPO, (0.Percusión, 1.Cuerdas, 2.Vientos, 3.Otra), String INSTRUMENTO
- Concierto: int DIA, int MES, String LUGAR, Músico V[32] (vector que reúne los datos de 32 músicos que forman parte del concierto

La clase concierto no pertenece a ninguna jerarquía

Se pide:

- Definir las tres clase e instancias de las mismas en el main().
- Definir e implementar constructores.
- Definir e implementar los métodos Cargar(parámetros necesarios) para las tres clases.
- Definir e implementar para la clase Concierto el método CuentaInstrimentos que recibe como parámetro un tipo de instrumento y retorna la cantidad de músicos cuyos instrumentos responden a dicho parámetro.
- Definir el método Mostrar() para la clase Músico e implementarlo desde el objeto de la clase Concierto antes declarado.
- Declarar un objeto Músico a partir de una Persona. Llamar a partir de él a la función Cargar.

33. Para calcular la bonificación navideña que un generoso empresario otorgará a cada uno de sus N empleados, se organizan los datos de una clase de la siguiente manera:

Un vector dinámico donde cada posición tiene la estructura:

Legajo (numérico)

Nombre (cadena de caracteres)

Categoría (numérico-0,1,2)

Sueldo básico

Años de antigüedad en la empresa

Un vector de porcentajes donde cada posición se corresponde con los posibles valores de categoría.

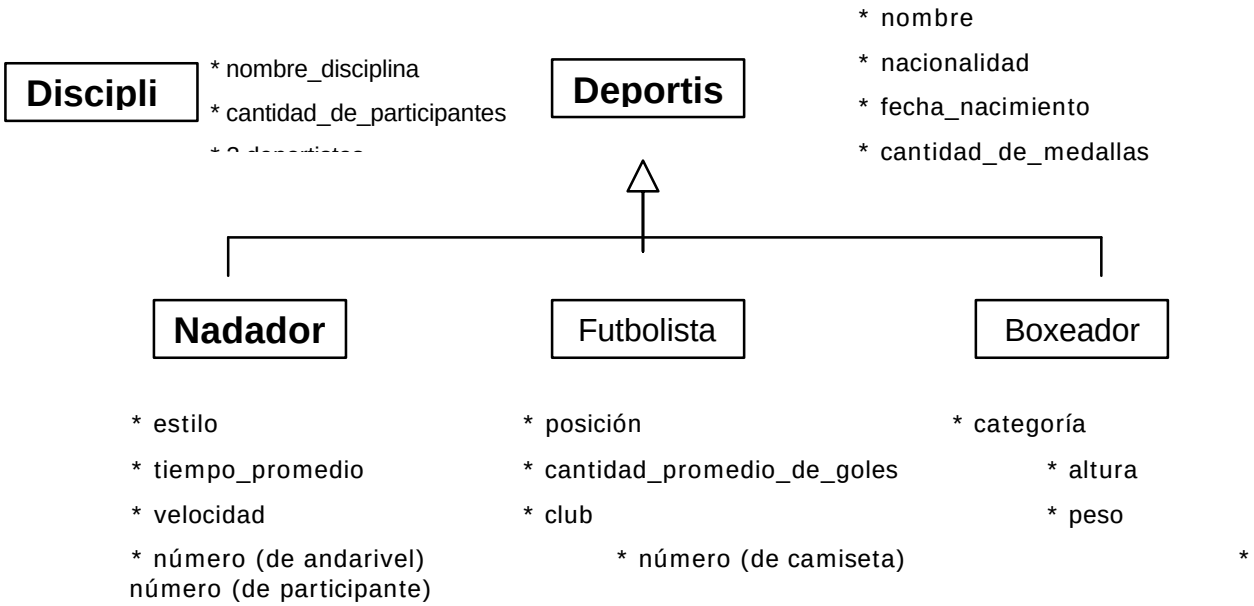
Se pide:

- Definir la clase e instancia de la misma en el main().
- Definir e implementar constructores.
- Definir e implementar el método OrdenAlfabético.
- Definir e implementar un método que permita acumular y retornar el monto total pagado por la empresa en concepto de sueldos básicos para sus empleados cuya categoría coincida con un valor de código pasado como parámetro.
- Mostrar por pantalla los datos de cada empleado destacando la bonificación a percibir, la cual se logra a través de la siguiente operativa:

  (porcentaje según categoría*sueldo)+(2*años de antigüedad)

Los porcentajes son: 10% para la categoría 0; 13% para la categoría 1; 15% para la categoría 2.

34. Se desean guardar los datos de los deportistas que obtendrán medallas en las olimpíadas que se están llevando a cabo. Para ello se utiliza la siguiente jerarquía de clases:



Además se cuenta con una clase **Disciplina** que contiene los siguientes datos:

- ✓ nombre_disciplina
- ✓ cantidad_de_participantes
- ✓ un vector de tres referencias a la clase **Deportista** donde el primero represente el nadador, futbolista o boxeador que ganó la medalla de oro en dicha disciplina, la segunda posición representa la medalla de plata y la tercera representa la de bronce.

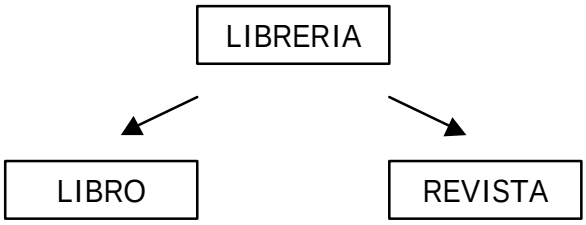
Se pide:

- Definir las clases y sus métodos correspondientes teniendo en cuenta que la clase Deportista nunca será instanciada.
- Definir constructores (con argumentos que tengan valores por defecto).
- Definir las funciones necesarias para mostrar todos los atributos de cada clase.
- Definir las funciones necesarias para modificar todos los atributos de cada clase.
- Definir en cada una de las 3 clases derivadas de Deportista un método que permita cargar el número correspondiente, realizando los controles pertinentes a cada deporte, a saber: el número de andarivel debe estar entre 1 y 8, el número de camiseta (fútbol) debe valer entre 1 y 11 y el número de boxeador debe estar entre 1 y 40 (cantidad máxima de participantes por categoría).
- Definir en Disciplina una función **entregar_medallas()** que pida por teclado los datos de cada uno de los 3 participantes que ganó una medalla en dicha disciplina y cree los objetos necesarios.
- Implementar en el main() un vector de 5 disciplinas y llamar a los métodos anteriormente definidos.

Nota:

- ✓ La clase Disciplina no pertenece a la jerarquía de herencia.
- ✓ Se pueden agregar más disciplinas si se desea, por ejemplo ciclismo, atletismo, básquetbol, etc. (No es obligatorio).
- ✓ La disciplina y los deportistas de la misma deben ser coincidentes; por ejemplo si la disciplina es natación los punteros de dicha disciplina deben apuntar a nadadores y no a futbolistas.
- ✓ En los deportes como fútbol, donde se juega en equipo, las medallas serán representadas por los capitanes del mismo.
- ✓ Todos los datos de tipo cadena de caracteres deben ser gestionados con punteros.

35. En un supermercado se tiene la siguiente jerarquía de productos:



Los datos de cada clase son los siguientes:

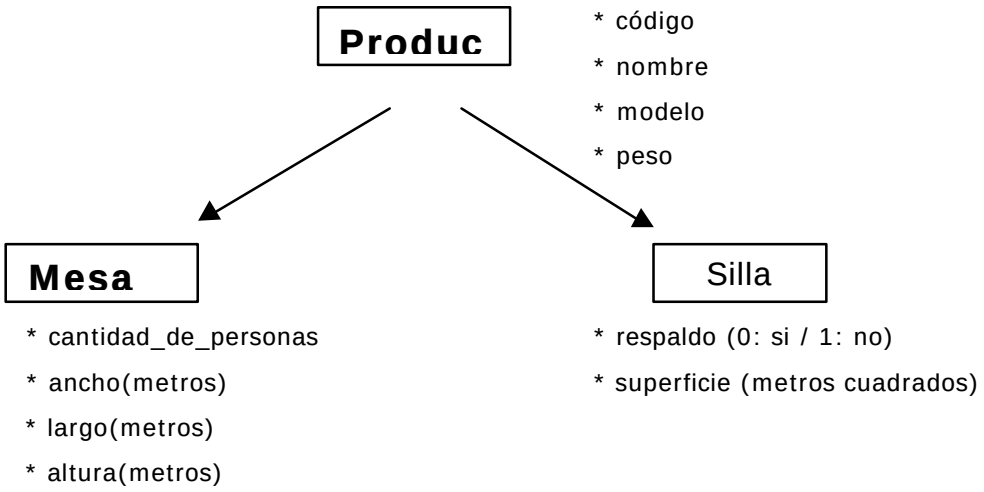
LIBRERÍA	LIBRO	REVISTA
Titulo	Autor	Numero
Editorial	Tomo	
Año edición	Edición	
Precio	ISBN	
Cantidad en stock		

Se pide:

1. Los constructores de las tres clases

2. Una función que muestre los datos de cada una de las distintas clases (debe actuar polimórficamente)
3. Implementar las clases en un arreglo unidimensional de 60 componentes con referencias a la clase base LIBRERÍA, donde cada componente referencia a un objeto de alguna de las dos clases derivadas.
4. A partir del vector calcular el valor total de stock que posee la librería

36. Una fábrica de muebles de la ciudad desea guardar la información sobre sus productos y la manera en que se lleva a cabo la fabricación de la siguiente forma. Para ello se utiliza la siguiente estructura:



1) Definir las tres clases y sus métodos correspondientes teniendo en cuenta que la clase Producto nunca será instanciada.
2) Definir constructores (con argumentos).
3) Definir en Producto: `mostrarDatos()` muestra los valores para todos los atributos de la instancia; `calcularCosto()` que retorna el costo de fabricación del producto; `ObtenerPeso()` que retorna el peso del producto; CalcularSuperficie() que retorna la superficie del producto.
4) Definir en Mesa: `mostrarDatos()` muestra los valores para todos los atributos de la instancia; `calcularCosto()` que retorna el costo de fabricación de la mesa (el costo es $20 por cada metro de superficie); CalcularSuperficie() que retorna la superficie de la mesa.
5) Definir en Silla: `mostrarDatos()` muestra los valores para todos los atributos de la instancia; `calcularCosto()` que retorna el costo de fabricación de la silla (el costo es $10 por cada metro de superficie); CalcularSuperficie() que retorna la superficie de la silla.
6) Definir una clase Plan_de_fabricación cuyos datos miembro sean:
   - Fecha de fabricación.
   - Cantidad de productos a fabricar
   - Un vector de 15 referencias de Producto.

   Y sus métodos: constructor copia; destructor; `agregarProducto()` que agrega al vector una silla o una mesa, preguntando cual es el tipo de producto que desea agregar; `mostrarDatos()` muestra los valores para todos los atributos de la instancia (incluyendo los datos de los elementos del vector); `calcularCosto()` que retorna el costo total del plan de fabricación; CalcularSuperficie() que retorna la superficie total a fabricar por dicho plan.

Todas estas funciones deben implementarse en el Main( ) mediante un objeto de tipo

Plan_de_fabricación.

37. En una empresa constructora de viviendas después del relevamiento realizado en la etapa de análisis se ha detectado la siguiente organización de los datos sobre los tipos de viviendas que la empresa construye. La misma ha solicitado a nuestra consultora el desarrollo de un software para el manejo y mantenimiento de la información.

Se detectaron distintos tipos de entidades, cada uno de ellos con sus cualidades particulares. La información recopilada en la etapa de análisis se indica a continuación:

- VIVIENDA: que posee los siguientes datos código vivienda, nomenclatura catastral, domicilio, valuación, superficie edificada, base imponible
- CASA: derivada de VIVIENDA, representa a aquellas viviendas de la empresa constructora que poseen además de los atributos de la clase base la superficie del terreno
- DEPARTAMENTO: derivada de VIVIENDA, los objetos de esta clase contendrán además de los atributos de VIVIENDA los siguientes atributos propios: piso, departamento

Se pide implementar en PPO lo siguiente:

1. Definir todas las clases teniendo en cuenta que la clase VIVIENDA nunca será instanciada.
2. Definir constructores (con parámetros por defecto).
3. Definir para la clase VIVIENDA el método "calcular_precio(cod_vivienda)" que calcula el valor de la vivienda que se obtiene de multiplicar la base imponible por la valuación dividido la superficie edificada.
4. Definir los métodos Cargar, Modificar y Mostrar para las clases que sean necesarias.
5. Definir en el main() un vector de 10 referencias a la clase base, en el cual los primeros 5 contengan objetos de tipo CASA y los 5 restantes DEPARTAMENTO. Luego crear objetos de las clases derivadas e implementar los métodos anteriormente definidos.

38. Para administrar los empleados la facultad, se debe desarrollar un programa orientado a objetos que pueda manejar mediante clases las siguientes entidades descubiertas durante la etapa de análisis del sistema:

- **Empleado**: es un empleado genérico que tiene las siguientes propiedades:

  -legajo

  -nombre

  -horas: cantidad de horas mensuales de trabajo del empleado

  y debe poseer las operaciones que se indican:

  -constructor con parámetros que inicialice todos los atributos

  -char * mostrarNombre() que retorna el nombre del empleado

  -float calcSueldo() que retorne el sueldo cobrado por el empleado

- **Docente**: empleado que dicta clases en algún curso de la facultad. Las características de este tipo de empleados incluyen todas las de la entidad anterior, más las siguientes propias:

  -antigüedad: cantidad de años que lleva al frente de un curso

  -valor_x_hora: cantidad de pesos que cobra por cada hora cátedra dictada

  -título: profesión en la que el docente se ha diplomado (ej: Ing., Dr., etc.)

  y debe poseer las siguientes operaciones:

  -constructor con parámetros que inicialice todos los atributos

  -char * mostrarNombre() que retorna el nombre del docente, precedido de su título

  -float calcSueldo() que calcule y retorne el sueldo neto cobrado por el docente. En este cálculo, al resultado de multiplicar el valor de la hora por la cantidad de horas se le debe sumar un

porcentaje igual a la cantidad de años de antigüedad

- **No_Docente**: empleado que se encarga de tareas administrativas. Las características de este tipo de empleados incluyen todas las de la clase Empleado, más las siguientes propias:

  -categoría: nivel jerárquico al que pertenece el empleado. Los posibles niveles y su sueldo por hora trabajada son: cat. 1 - $13,25; cat. 2 - $11,50; cat. 3 - $9,75

  -área: departamento al que pertenece el empleado (ej: decanato, sistemas, ciencias. básicas, etc.)

  y debe poseer las siguientes operaciones:

  -constructor con parámetros que inicialice todos los atributos

  -float calcSueldo() que calcule y retorne el sueldo neto cobrado por el empleado no docente

- **Facultad**: que puede tener hasta 50 empleados y contiene los siguientes atributos:

  -nombre: denominación de la facultad (ej: Facultad Regional Córdoba)

  -dirección: domicilio de la facultad

  -empleados: vector de 50 referencias a Empleado

  y debe poseer las siguientes operaciones:

  -constructor con parámetros que inicialice los atributos (sólo inicializa atributos, no carga empleados).

  -int agregarEmpleado(Empleado & e): agrega el empleado recibido al vector de empleados, si quedaba alguna posición libre. Devuelve 1 si pudo agregarlo o 0 si no pudo

  -void mostrarDatos(): muestra por pantalla los valores de los atributos propios y luego los nombres junto al sueldo de cada empleado cargado

Hacer un pequeño main() donde se ejemplifique el uso de las clases, creando una instancia de Facultad y agregándole algunos empleados.

39. Una empresa telefónica desea guardar los datos de sus clientes utilizando la siguiente estructura:

- Clase Teléfono con los atributos String nomTitular, int numeroTel, char tipo ('f' = fijo y 'c' =celular), static valorPulso y static valorMinuto (inicializándose estos 2 últimos con los valores 0.23 y 0.33 respectivamente)
- Clase TeléfonoFijo que deriva de Teléfono con los siguientes atributos propios: String dirección e int cantPulsos.
- Clase Celular que deriva de Teléfono con los atributos int codPlan (1, 2 ó 3) e int cantMinutos.

<u>Se pide:</u>

1) Definir las tres Clases y los Métodos correspondientes, teniendo en cuenta que no se instanciará la clase Teléfono.
2) Definir constructor sin parámetros para la clase Teléfono y con parámetros para TeléfonoFijo y Celular.
3) Definir el método cargar() para todas las clases.
4) Definir el método devolverTipo() para la clase Teléfono.
5) Definir el método monto() que devuelva el total a pagar por el cliente, que se calcula para:

   TeléfonoFijo: cantPulsos * valorPulso

   Celular : cantMinutos * valorMinuto + (15, 22 ó 34 según el codPlan sea 1, 2 ó 3)

Realizar un main() que contenga un vector de 100 referencias a Teléfono. El usuario elegirá mediante un pequeño menú si crea objetos de TeléfonoFijo o Celular. Luego de cargados los objetos, determinar el monto **total** a pagar de los teléfonos **celulares**.

40. Nos han pedido que desarrollemos un pequeño programa para administrar las fichas de los distintos objetos expuestos en el museo Rocksen. Estos objetos pueden ser de 2 tipos: históricos

(piedras, fósiles, plantas, momias, etc.) u obras de arte (esculturas, pinturas, etc.). Durante la etapa de análisis del sistema se descubrieron las siguientes entidades:

- **Objeto**: es cualquier elemento expuesto, que posee las siguientes propiedades:

  -código

  -nombre

  -año: en que fue realizada la obra o estimación de formación/creación de elementos naturales

  y debe poseer las operaciones que se indican:

  -constructor con parámetros que inicialice todos los atributos

  -String mostrarNombre() que retorna el nombre del objeto

  -String getDatos() que retorna todos los datos del objeto en exposición

- **Histórico**: elemento antiguo expuesto. Las características de este tipo de objetos incluyen todas las de la entidad anterior, más las siguientes propias:

  -zona: cadena de caracteres que indica la zona geográfica en la que fue encontrado

  -origen: cadena de caracteres que describe el origen del objeto

  y debe poseer las siguientes operaciones:

  -constructor con parámetros que inicialice todos los atributos

  -String mostrarNombre() que retorna el nombre del objeto, seguido por la zona, ej: león (África).

  -String getDatos()que retorna todos los datos del objeto en exposición

  -void mostrarDatos()que muestra por pantalla todos los datos del objeto en exposición

- **Obra**: obra de arte expuesta. Las características de este tipo de objetos incluyen todas las de la clase Objeto, más las siguientes propias:

  -autor: nombre del autor de la obra

  -país: país de origen de la obra

  y debe poseer las siguientes operaciones:

  -constructor con parámetros que inicialice todos los atributos

  -String mostrarNombre()retorna el nombre de la obra, seguido por su autor, ej: Girasoles (Van Gogh).

  -String getDatos()que retorna todos los datos de la obra de arte

  -void mostrarDatos()que muestra por pantalla todos los datos de la obra de arte

- **Sala**: sala de exposiciones que puede tener hasta 100 objetos de cualquier tipo en exposición y contiene los siguientes atributos:

  -número

  -ala: ala del museo donde se ubica la sala (Norte o Sur)

  -objetos: vector de 100 referencias a Objeto para los elementos en exposición

  y debe poseer las siguientes operaciones:

  -constructor con parámetros que inicialice los atributos (sólo inicializa atributos, no carga objetos)

  -int agregarObjeto(Objeto & o): agrega el objeto recibido al vector de objetos, si quedaba alguna posición libre. Devuelve 1 si pudo agregarlo o 0 si no pudo

  -void mostrarDatos(): muestra por pantalla los valores de los atributos propios y luego los datos completos de cada objeto cargado

Desarrollar un pequeño main() donde se ejemplifique el uso de las clases, creando una instancia de Sala y agregándole algunos objetos.

41. Un instituto de belleza desea organizar la información que utiliza diariamente a través de un pequeño sistema que le permita realizar la facturación de sus clientes. El mismo otorga servicios de belleza a sus clientes como peinado, corte, manicura, etc; y también vende sus productos a los mismos como shampoo, baño de crema, cera depiladora, etc. Por consiguiente ha decidido organizar la información a través de las siguientes clases:

- `Artículo`: que posee como atributos código, descripción y precio; y sus métodos son: constructor por defecto; `cargarDatos()` carga valores para todos los atributos de la instancia; `calcularGanancia()` que retorna la ganancia neta que proporciona dicho artículo al instituto; `ObtenerPrecio()` que retorna el precio del artículo; ConsultarCantidad() que retorna la cantidad de artículos creados hasta el momento. Se debe tener en cuenta que el código de artículo es un valor que no se ingresa por teclado sino que es generado por el sistema en el momento de crear el artículo.
- `Producto`: clase derivada de `Artículo`, posee como atributos caducidad (cantidad de días), cantidad en stock; y sus métodos son: constructor por defecto; `cargarDatos()` carga valores para todos los atributos de la instancia; `calcularGanancia()` que retorna la ganancia neta que proporciona dicho producto al instituto (la ganancia es el 30% del precio del producto); ConsultarStock() que retorna la cantidad en stock de ese producto.
- `Servicio`: clase derivada de `Artículo`, posee como atributo tiempo (en horas); y sus métodos son: constructor por defecto; `cargarDatos()` carga valores para todos los atributos de la instancia; `calcularGanancia()` que retorna la ganancia neta que proporciona dicho servicio al instituto (la ganancia es el 50% del precio del producto menos $ 0.50 por cada hora utilizada); ClasificarServicio() que retorna 1 si el servicio se lleva a cabo en menos de una hora, retorna 2 si el servicio se lleva a cabo en un tiempo de 1 a 3 horas y retorna 3 en el resto de los casos.
- `Factura`: clase que posee como atributos número de factura, nombre del cliente y artículos (vector de 10 punteros a la clase `Artículo` que permiten manejar hasta 10 productos y/o servicios) ; y sus métodos son: constructor con argumentos; destructor; `agregarArtículo()` que agrega al vector un artículo, preguntando cual es el tipo de artículo que desea agregar (producto o servicio) quitar`Artículo(int código)` que elimina si es posible un artículo de la factura según el código pasado por parámetro; `calcularPrecio()` que retorna el precio total de la factura; `calcularGanancia()` que retorna la ganancia total de la factura.

Crear en el main() un ejemplo de factura y llamar a los métodos anteriormente desarrollados.

42. Una empresa consultora de sistemas organiza el trabajo dentro de la misma de acuerdo a grupos de la siguiente manera. Cada vez que se presenta un proyecto se conforma un grupo encargado de desarrollar el mismo; este grupo está formado por un jefe de proyecto y un conjunto de analistas cuyo número puede variar de 3 a 6 personas. Para organizar dicha información se tiene la siguiente jerarquía de clases:

- Analista: que posee como atributos código, nombre, sueldo y tarea (que desempeña dentro del grupo).
- `Jefe`: clase que agrega a los atributos de Analista estas características propias: porcentaje (que obtiene por el proyecto) y varios (1-está a cargo de más de un proyecto a la vez, 2- no está a cargo de más de un proyecto a la vez).
- Proyecto: clase que posee como atributos código de cliente, fecha de inicio, monto, jefe de proyecto (instancia de la clase Jefe) y miembros (vector de 6 punteros a la clase Analista que permiten incluir hasta 6 Analistas en el proyecto).

Para cada una de estas clases se deberán definir sólo los métodos solicitados, agregando todos los elementos que crea necesarios para cumplir con este objetivo. Al final, se deberá desarrollar un pequeño `main()` ejemplificando su uso.

Se pide:

a. Para la clase Analista definir los siguientes métodos: constructor con argumentos; `mostrarDatos()`, muestra los valores de todos los atributos de la instancia; `tomarSueldo()` que retorna el sueldo de dicho empleado.

b. Para la clase Jefe definir los siguientes métodos: constructor con argumentos; `mostrarDatos()`, muestra los valores de todos los atributos de la instancia;

`tomarSueldo()`que retorna el sueldo del jefe (calculado como su sueldo neto más el porcentaje correspondiente por el proyecto que está ejecutando).

c. Para la clase Proyecto definir los siguientes métodos: constructor por defecto; `cargarDatos()`, carga los datos de todos los atributos de la instancia; `mostrarDatos()`, muestra los valores de todos los atributos de la instancia; calcularCosto() que retorna el costo del proyecto de acuerdo a los sueldos de cada uno de los sueldos de sus miembros; agregarAnalista() que agrega un analista al vector miembros.

43. Debemos desarrollar un sistema para administrar las ventas de un importante portal de Internet que se dedica a la comercialización de libros y CDs. Los pedidos son realizados en línea y pueden contar de un máximo de 5 productos, que luego serán enviados al cliente a través de un servicio de correo. Durante el análisis realizado al sistema se distinguieron las siguientes clases:

- `Producto`: que posee como atributos código, descripción y precio.
- `Libro`: clase que agrega a los atributos de `Producto` estas características propias: nombre del autor, cantidad de páginas y encuadernación (1-económica, 2-tapas duras).
- `CD`: clase que agrega a los atributos de `Producto` las siguientes características propias: nombre del artista, cantidad de temas y presentación (1-simple, 2-doble).
- `Pedido`: clase que posee como atributos número de pedido, nombre del cliente y productos (vector de 5 referencias a la clase `Producto` que permiten manejar hasta 5 libros y/o CDs).

Para cada una de estas clases se deberán definir sólo los métodos solicitados, agregando todos los elementos que crea necesarios para cumplir con este objetivo. Al final, se deberá desarrollar un pequeño `main()` ejemplificando su uso. **Se pide**:

a. Para la clase **`Producto`** definir los siguientes métodos: constructor copia; `cargarDatos()` carga valores para todos los atributos de la instancia; `calcularPeso()` que retorna el peso neto de dicho producto; `getPrecio()` que retorna el precio del producto.

b. Para la clase **`Libro`** definir los siguientes métodos: constructor copia; `cargarDatos()` carga valores para todos los atributos de la instancia; `calcularPeso()` que retorna el peso neto de dicho producto, calculado como la cantidad de hojas multiplicada por 10gr, a lo que se le suman 100gr cuando se trata de un libro de tapas duras.

c. Para la clase **`CD`** definir los siguientes métodos: constructor copia; `cargarDatos()` carga valores para todos los atributos de la instancia; `calcularPeso()` que retorna el peso neto de dicho producto, considerando que <u>cada</u> disco pesa 150gr.

d. Para la clase **`Pedido`** definir los siguiente métodos: constructor por defecto; `agregarProducto(int tipo)` que agrega al vector un producto, de acuerdo al `tipo` especificado (1: `Libro`, 2: `CD`); `calcularPeso()` que retorna el peso total del pedido; `calcularPrecio()` que retorna el precio total del envío, sumándole al precio de los productos $5 por cada 500gr en concepto de gastos de envío.

44. En un taller mecanico se desea desarrollar un sistema que administre la informacion los moviles que se reparan en el mismo. Los moviles a tratar pueden ser autos o motos. Las reparaciones las lleva a cabo un encargado y no puede tener mas de 7 moviles en reparacion en un mismo momento. Durante el análisis realizado al sistema se distinguieron las siguientes clases:

- `Movil`: que posee como atributos código, falla y precio de la reparacion.
- `Auto`: clase que agrega a los atributos de `Movil` estas características propias: marca del auto, ancho , largo y tipo (1-base, 2-full).
- `Moto`: clase que agrega a los atributos de `Movil` las siguientes características propias: marca de la moto, velocidad maxima y marchas (1-si, 2-no).
- `Taller`: clase que posee como atributos direccion, nombre del encargado y reparaciones (vector de 7 punteros a la clase `Movil` que permiten manejar hasta 7 autos y/o motos).

Para cada una de estas clases se deberán definir sólo los métodos solicitados, agregando todos los elementos que crea necesarios para cumplir con este objetivo. Al final, se deberá desarrollar un

pequeño `main()` ejemplificando su uso. **Se pide**:

a. Para la clase **`Movil`** definir los siguientes métodos: constructor con argumentos; `mostrarDatos()`, muestra los valores de todos los atributos de la instancia; `tomarFalla()` que retorna la Falla de dicho movil; `calcularPrecio()` que retorna el precio de reparacion del movil.

b. Para la clase **`Auto`** definir los siguientes métodos: constructor con argumentos; `mostrarDatos()`, muestra los valores de todos los atributos de la instancia; `tomarFalla()`que retorna la Falla de dicho movil; `calcularPrecio()` que retorna el precio de reparacion del movil, teniendo en cuenta que si el movil es un auto se incrementa el precio en un 15%.

c. Para la clase **`Moto`** definir los siguientes métodos: constructor con argumentos; `mostrarDatos()`, muestra los valores de todos los atributos de la instancia; `tomarFalla()`que retorna la Falla de dicho movil; `calcularPrecio()` que retorna el precio de reparacion del movil, teniendo en cuenta que si el movil es una moto se incrementa el precio en un 5%.

d. Para la clase **`Taller`** definir los siguiente métodos: constructor por defecto; `agregarMovil(int tipo)` que agrega al vector un movil, de acuerdo al `tipo` especificado (1: `Auto`, 2: `Moto`); `TomarFalla()` que retorna un vector de 7 fallas, una correspondiente a cada movil; `calcularPrecio()` que retorna el precio total de las reparaciones, sumándole al precio de los moviles $5 por cada movil que posee el vector.

45. Los dueños de un nuevo Centro Comercial nos solicitan que desarrollemos un programa que les permita administrar los locales que van siendo vendidos o alquilados a los locatarios. Dicho centro cuenta con 20 locales para instalar negocios de cualquier rubro, los que se pueden alquilar o vender; es decir que algunos locales estarán alquilados y otros vendidos (todos estos deben ser administrados por nuestro sistema) y el resto estará disponible para realizar una transacción.

Durante el análisis realizado se distinguieron las siguientes clases:

- Local: que posee como atributos número de local (de 1 a 20), metros cuadrados, rubro (del negocio instalado) y monto de expensas a pagar.
- Vendido: clase que agrega a los atributos de Local las siguientes características propias: nombre del dueño, persona a cargo y precio del local.
- Alquilado: clase que agrega a los atributos de Local las siguientes características propias: nombre del inquilino, costo anual del alquiler y año de finalización del contrato.
- Centro Comercial: clase que posee como atributos nombre (del centro), dueño, dirección y locales (vector de 20 punteros a la clase Local que únicamente poseerá objetos en las posiciones correspondientes a los locales vendidos o alquilados).

Para cada una de estas clases se deberán definir sólo los métodos solicitados, agregando todos los elementos que crea necesarios para cumplir con este objetivo.

Se pide:

a. Para la clase **`Local`** definir los siguientes métodos: constructor copia; `cargarDatos()` carga valores para todos los atributos de la instancia; `calcularPrecio()` retorna el monto que se le debe pagar al Centro Comercial.

b. Para la clase **`Vendido`** definir los siguientes métodos: constructor copia; `cargarDatos()` carga valores para todos los atributos de la instancia; `calcularPrecio()` retorna el monto que se le debe pagar al Centro Comercial como resultado de sumarle las expensas al precio del local.

c. Para la clase **`Alquilado`** definir los siguientes métodos: constructor copia; `cargarDatos()` carga valores para todos los atributos de la instancia; `calcularPrecio()` retorna el monto que se le debe pagar al Centro Comercial como resultado de sumarle a las expensas una doceava parte del costo anual de alquiler.

d. Para la clase **`CentroComercial`** definir los siguiente métodos: constructor por defecto, `agregarLocal(int nro, int tipo)` que agrega el local indicado por el parámetro `nro` al vector de locales, de acuerdo al `tipo` especificado (1: local vendido, 2: local alquilado); `calcularValor()` que retorna el valor total recaudado por el Centro Comercial en el primer mes en concepto de ventas y alquileres de locales.

46. Se desea organizar la comercialización de entradas para un partido de fútbol. Para este evento se venden 2 tipos de entradas: general y platea. Para controlar la venta, cada boletería puede tener hasta 100 entradas, entre generales y plateas. Durante el análisis realizado al sistema se distinguieron las siguientes clases:

- `General`: que posee como atributos número de serie, precio y puerta (número de la puerta de ingreso).
- `Platea`: clase que agrega a los atributos de `General` estas características propias: adicional (monto extra que se debe abonar sobre el precio de una entrada general), sector (zona de la platea a la que se puede acceder con esa entrada, un caracter) y número de asiento.
- `Boletería`: clase que posee como atributos nombre del vendedor y entradas (vector de 100 referencias a la clase `General` que permiten manejar hasta 100 entradas generales y/o plateas).

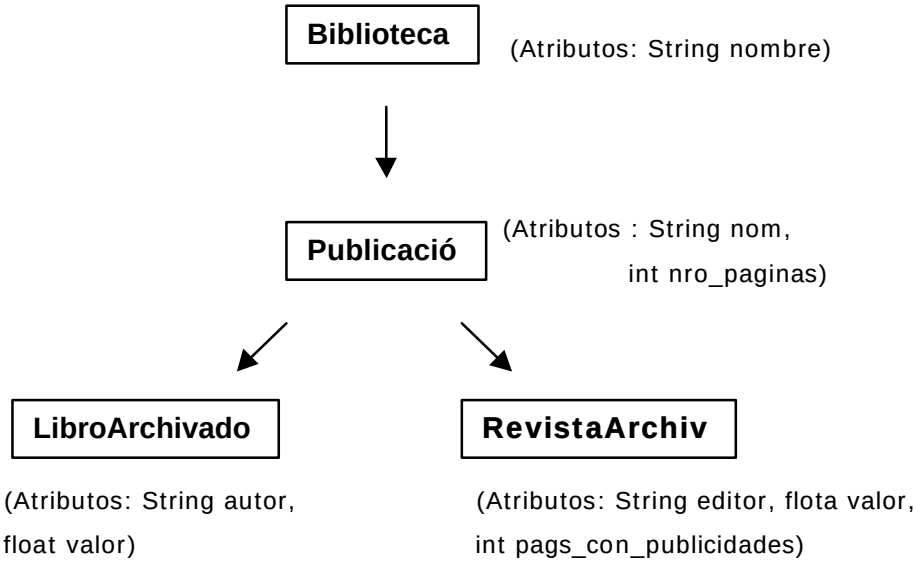
Para cada una de estas clases se deberán definir sólo los métodos solicitados, agregando todos los elementos que crea necesarios para cumplir con este objetivo. Al final, se deberá desarrollar un pequeño `main()` ejemplificando su uso. **Se pide**:

a. Para la clase **`General`** definir los siguientes métodos: constructor con parámetros; `mostrarDatos()`, muestra los valores de todos los atributos de la instancia; `getPrecio()` que retorna el precio de la entrada.

b. Para la clase **`Platea`** definir los siguientes métodos: constructor con parámetros; `mostrarDatos()`, muestra los valores de todos los atributos de la instancia; `getPrecio()` que retorna el precio de la entrada, considerando que la Platea le agrega un adicional al precio de la General.

c. Para la clase **`Boletería`** definir los siguiente métodos: constructor con parámetros, que inicializa el vector de entradas dejando 60 Generales y 40 Plateas; `venderEntrada(int tipo)` que elimina del vector la entrada indicada de acuerdo al `tipo` especificado (`1`: `General`, `2`: `Platea`).

47. Cierta biblioteca barrial organiza su información (y los procesos necesarios para utilizarla) según la siguiente jerarquía de clases:

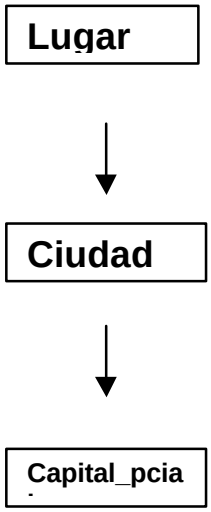


Definir los métodos:

1. Cconstructor de todas las clases.
2. "void mostrar()" en todas las clases redefiniendo en cada una de ellas.
3. "int pags_sin_publicidad()" retorna la cantidad de páginas de una RevistaArchivada que carece de publicidades.

4. Declarar en el main() un vector de 50 referencias a la clase Biblioteca, pero sabiendo que las posiciones pares representan a libros archivados y las impares a revistas. Implementar desde todos sus componentes el método mostrar e imprimir la cantidad de paginas sin publicidades sólo para impares.

48 Una serie de datos se organizan en las siguientes clases para representar la distribución geográfica de un país:



Cada una de éstas clases se componen así:

- lugar = String nom
  char tipolat //valores 'N' o 'S'
  float latitud
  char tipolon //valores 'E' u 'O'
  flota longitud

- ciudad = String intte //intendente
  int codprov

- capital_pcial = String gobern

Se pide:

- ✓ Definir las tres Clases.
- ✓ Definir e implementar constructores.
- ✓ Definir e implementar el método mostrar() para todas las clases (teniendo en cuenta que luego será accedido con punteros).
- ✓ Definir e implementar un método en "lugar" que reciba como parámetro un tipo y un valor de latitud y modifique los valores existentes.
- ✓ Declarar en el main() un objeto de cada clase e implementar los métodos definidos. Declarar además un vector de 6 referencias a lugar y crear en forma dinámoca objetos de tipo lugar para las 2 primeras posiciones, de tipo ciudad para las 2 siguientes, etc. Por último implementar los métodos con el vector.

49. En el sector de internación de una clínica se lleva el registro de los pacientes internados, de los cuales se tiene la siguiente información: número de historia clínica, datos del paciente, habitación asignada, tipo de terapia (normal o intensiva), fecha de ingreso y fecha de salida. De cada paciente se registra: nombre, documento, domicilio, teléfono y edad. Para realizar consultas o actualizaciones se trabajará con una lista o vector.

Definir las clase necesarias, constructores y funciones para realizar lo que se les pida en cada punto y utilizarlas en el main().

- ✓ Crearlista(): carga en la lista aquellos pacientes que se encuentran en terapia intensiva.
- ✓ Salpac(hiscli,salida): registra la fecha de salida de un paciente determinado.
- ✓ Canpac(): determina la cantidad de pacientes en terapia intensiva.
- ✓ Listarpacientes(): listar los datos de todos los pacientes que estén en la lista.

50. La dirigencia de un club deportivo y cultural nos encargó la realización de un sistema que pueda mantener todos los datos de sus socios:

En un relevamiento se detectó que los socios se encuentran divididos en dos grupos: SociosNoJugadores (socios que no integran equipos de deportes) y SociosJugadores (socios que integran equipos de deportes).

Se deben manejar para ambos tipos de socios los datos personales: Nombre, Apellido, Dirección, Teléfono, Precio_cuota_social, Nro_de_cuotas_atrasadas, Edad y Ocupación; y solo para los socios jugadores Deporte_que_practica y Posición.

Nos interesa que tanto los socios como los socios no jugadores puedan Mostrar todos sus atributos, Retornar el monto total de deuda (teniendo en cuenta que los socios jugadores pagan un 10% más que los socios no jugadores).

Éstos dos tipos de socios deben estar coordinados desde una clase club que pueda mantener hasta 200 socios, Nombre_del_club, Dirección y Teléfono. A ésta clase se le debe agregar el comportamiento necesario para que Muestre todos los datos de los socios junto con su deuda, Muestre el total adeudado de todos los socios y pueda afiliar a un Nuevo socio.

Todas las clases deberán tener un constructor con parámetros y un destructor.

Se debe implementar un pequeño main() ejemplificando el uso de las clases antes descriptas.