Universidad Politécnica de Madrid

Programación Orientada a Objetos con Java

Área de Documentación del MIW

http://miw.eui.upm.es

Título	Programación Orientada a Objetos con Java (POOJ)		
Referencia	Java EE		
Autor	Jesús Bernal Bermúdez (j.bernal@upm.es)		
Colaborador	Luis Fernández Muñoz (jluis.fernandezm@upm.es)		
Versión	1.1	**Fecha**	01/09/2012

Versión	Fecha	Descripción
1.0	20/12/2010	Versión inicial. Aproximación práctica y desde cero a la Programación Orientada a Objetos con Java
1.1	01/09/2012	Adaptación del formato a la documentación del MIW. Corrección de erratas y mejora de los ejemplos

Propósito
El propósito de este documento es asentar los conocimientos necesarios sobre la Programación Orientado a Objetos con Java, para poder seguir con normalidad el Máster en Ingeniería Web. Se considera una lectura altamente recomendable.

Audiencia
Futuros alumnos del Máster en Ingeniería Web, por la Universidad Politécnica de Madrid.

Conocimientos previos
Conocimientos de la Ingeniería en Informática, y especialmente en programación.

Referencias
http://docs.oracle.com/javase/tutorial/

Índice

1. Programación Orientada a Objetos....................7

Conceptos de la POO 7

Clases y Objetos *7*

Entorno de desarrollo Java 8

La consola *9*

El primer programa *10*

Ficharos JAR *10*

Clase IO *11*

Lenguaje de programación Java 12

Aspectos generales de la gramática *12*

Sentencias *16*

Sentencias condicionales *16*

Sentencias de bucles *17*

Clases y objetos 19

Clases: atributos y métodos *19*

Atributos *19*

Métodos *20*

Objetos *22*

Mensajes *24*

Elementos estáticos *26*

Clases de Java fundamentales *27*

Javadoc *28*

Herencia y polimorfismo 32

Herencia *32*

Clases abstractas *35*

Interfaces *37*

Ejercicios *37*

Colaboración entre clases 38

Relación de clases *39*

Beneficio de la herencia *40*

Beneficio de los interfaces *41*

Paquetes *42*

Visibilidad *43*

Método de Desarrollo de Programas 44

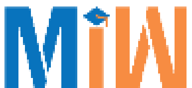
2. Programación avanzada 51

Excepciones 51

Manejo de excepciones *52*

Lanzamiento de excepciones *53*

Tipos genéricos 54
Enumerados 56
Colecciones de datos 57
Collection e Iterator *57*
List *58*
Set *59*
Queue *60*
SortedSet *60*
Map *60*
SortedMap *61*
Programación concurrente 61
Principios conceptuales *61*
Exclusión y sincronización *64*
3. Flujos de datos 69
Clases básicas 69
Clases InputStream y OutputStream *69*
Clases DataInputStream y DataOutputStream *70*
Clases ObjectInputStream y ObjectOutputStream *71*
Clases BufferedInputStream y BufferedOutputStream *73*
Clases InputStreamReader y OutputStreamWriter *73*
Ficheros y Directorios 74
Clases FileInputStream y FileOutputStream *74*
Clase File *74*
RandomAccessFile *76*
FileDialog *76*
URL 77



1. Programación Orientada a Objetos

Conceptos de la POO

La programación orientada a objetos establece un equilibrio entre la importancia de los procesos y los datos, mostrando un enfoque más cercano al pensamiento del ser humano. Se introduce un aspecto novedoso respecto al anterior paradigma: la herencia, facilitando el crecimiento y la mantenibilidad.

Las bases de la programación orientada a objetos son: abstracción, encapsulación, modularidad y jerarquización.

La ***abstracción*** es un proceso mental de extracción de las características esenciales, ignorando los detalles superfluos. Resulta ser muy subjetiva dependiendo del interés del observador, permitiendo abstracciones muy diferentes de la misma realidad

La ***encapsulación*** es ocultar los detalles que dan soporte a un conjunto de características esenciales de una abstracción. Existirán dos partes, una visible que todos tienen acceso y se aporta la funcionalidad, y una oculta que implementa los detalles internos.

La ***modularidad*** es descomponer un sistema en un conjunto de partes. Aparecen dos conceptos muy importantes: acoplamiento y cohesión.

- El acoplamiento entre dos módulos mide el nivel de asociación entre ellos; nos interesa buscar módulos poco acoplados
- La cohesión de un módulo mide el grado de conectividad entre los elementos que los forman; nos interesa buscar una cohesión alta

La ***jerarquía*** es un proceso de estructuración de varios elementos por niveles.

La programación orientada a objetos implementa estos cuatro conceptos con los siguientes elementos: clases y objetos, atributos y estado, métodos y mensajes, herencia y polimorfismo.

Clases y Objetos

Una clase describe las estructuras de datos que lo forman y las funciones asociadas con él. Una clase es un modelo con el que se construyen los objetos.

Un objeto es un ejemplar concreto de una clase, que se estructura y comporta según se definió en la clase, pero su estado es particular e independiente del resto de ejemplares. Al proceso de crear un objeto se le llama generalmente ***instanciar*** una clase.

Las clases asumen el principio de encapsulación, se describe una vista pública que representa la funcionalidad de la misma, y una vista privada que describe los detalles de implementación.

Una clase es el único bloque de construcción, y por lo tanto, en una aplicación Java sólo hay clases; no existen datos sueltos ni procedimientos.

ATRIBUTOS Y ESTADO

Un atributo es cada uno de los datos de una clase que la describen; no incluyen los datos auxiliares utilizados para una implementación concreta.

El estado de un objeto es el conjunto de valores de sus atributos en un instante dado.

MÉTODOS Y MENSAJES

Un método define una operación sobre un objeto. En general, realizan dos posibles acciones: consultar el estado del objeto o modificarlo. Los métodos disponen de parámetros que permiten delimitar la acción del mismo.

Nos podemos encontrar con diversos tipos de métodos:

- Consultan o modifican un atributo, normalmente nos referenciaremos a ellos como: getters & setters
- Realizan operaciones sobre el conjunto de atributos, calculando valores o realizando modificaciones
- Inicializan los atributos al principio del ciclo de vida, o liberan los recursos al final del ciclo; nos referiremos a ellos como constructores o destructores

Un mensaje es la invocación de un método de un objeto. Podemos decir que un objeto lanza un mensaje (quien realiza la invocación) y otro lo recibe (el que ejecuta el método).

Podemos rescribir que una ***clase*** es la descripción e implementación de un conjunto de atributos y métodos.

HERENCIA Y POLIMORFISMO

La herencia es una característica que permite a las clases definirse a partir de otras, y así reutilizar su funcionalidad. A la clase padre se le llama superclase, clase base..., y a la hija subclase, clase derivada....

El polimorfismo es la capacidad de que un mismo mensaje funcione con diferentes objetos. Es aquél en el que el código no incluye ningún tipo de especificación sobre el tipo concreto de objetos sobre el que se trabaja. El método opera sobre un conjunto de posibles objetos compatibles.

Entorno de desarrollo Java

El lenguaje Java fue creado por *Sun Microsystems Inc.*, Aparece en el año 1995 y debe, en gran medida, su popularidad al éxito del servicio WWW. Se creó en su origen para que fuese un lenguaje multiplataforma. Para ello se compila en un código intermedio: *bytecode* y necesita de una máquina virtual que lo ejecute. Normalmente, no utiliza código nativo, es decir, no se puede ejecutar directamente por el procesador.

Se disponen de varias plataformas Java para el desarrollo. Una plataforma es una combinación de hardware y software, usada para desarrollar y/o ejecutar programas.

Se va a empezar con el editor. Uno sencillo de libre distribución es el Notepad++:

http://notepad-plus.sourceforge.net/es/site.htm

Es un editor básico que reconoce la gramática del lenguaje; es recomendable para empezar con aplicaciones pequeñas, pero no para producir con efectividad.

Para el desarrollo y compilación de aplicaciones Java, utilizaremos: *Standard Edition* (Java SE) o *Java Development Kit* (JDK) de Sun:

http://java.sun.com/javase/downloads/index.jsp

Se trabaja mediante comandos en consola. Incluye, entre otras, las siguientes utilidades:

- El compilador: javac.exe. Un ejemplo: `javac Fich.java`
- El intérprete: java.exe. Ej. `java Fich`
- Un compresor: jar.exe. Ej. `jar –cvf fich.jar Uno.class Dos.class`
- El generador de documentación: javadoc.exe. Ej. `javadoc *.java`
- Un analizador de clases: javap.exe. Ej. `javap Fich`

Posteriormente, se verán plataformas más avanzadas como *Eclipse* o *Netbeans*.

Para la ejecución de aplicaciones Java (*Delivery Platforms*) es el *Java Runtime Environment* (JRE). Se instala automáticamente con Java SE.

Una vez instalado, debemos configurar la variable de entorno *PATH* para poder utilizar las utilidades desde la consola. Se debe añadir la ruta de la carpeta *bin*. En una instalación estándar podría ser:

```
C:\Program Files\Java\jdk1.6.0_16\bin;
```

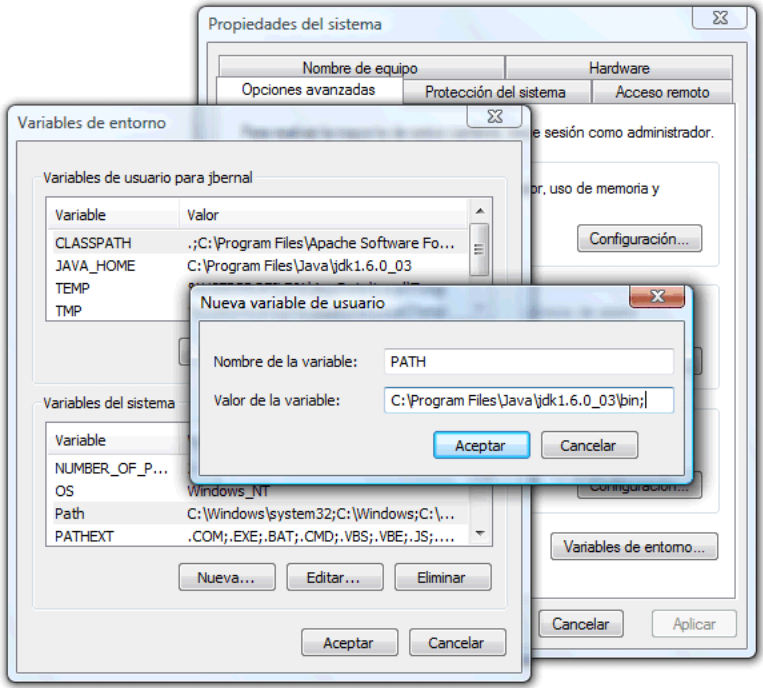


Figura 1. Java SE: variable de entorno PATH

La variable de entorno *CLASSPATH* tiene especial importancia. Si no está definida busca las clases en la carpeta donde se ejecuta el *.class. Si se define se debe incluir, además de los ficheros *.jar de otras librerías, la ruta de la carpeta actual ("./"). Un ejemplo genérico sería:

```
CLASSPATH:  .; c:\...\lib1.jar; ...\lib2.jar;
```

Atención: Para que tengan efecto los cambios en las variables de usuario hay que cerrar la consola o sesión.

La consola

La consola es una ventana (llamada *Símbolo del Sistema* en Windows) que nos proporciona un punto de entrada para escribir comandos. Estos comandos nos permiten realizar tareas sin utilizar un entorno gráfico.

Aunque resulte un tanto incómodo, es una buena alternativa para la primera aproximación a Java.

Cuando se inicia la consola, aparece la ruta de una carpeta y acaba con el símbolo ">". Cualquier comando Java lo deberemos ejecutar sobre la carpeta de trabajo.

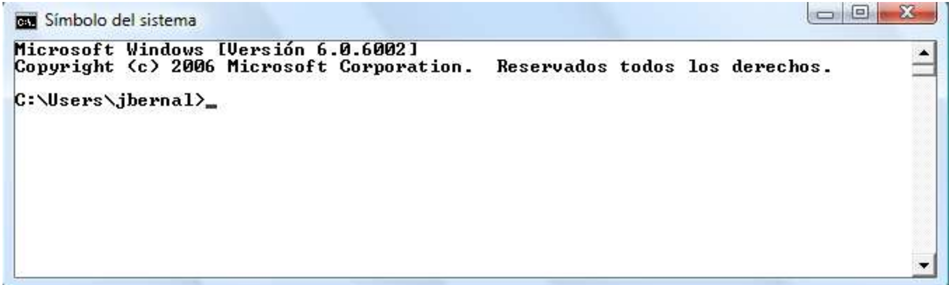


Figura 2. Java SE: consola

Sin profundizar en los diferentes comandos, veremos lo más básicos para poder realizar el trabajo con Java:

- Para cambiar de unidad, escribiremos la letra de unidad seguida de dos puntos, por ejemplo: "`g:`"
- `cd`. Con este comando cambiaremos de carpeta. Para subir un nivel escribiremos: "`cd..`", y para profundizar "`cd carpeta`"
- `dir`. Lista el contenido de la carpeta
- `type`. Visualiza el contenido de un fichero de texto: "`type fich.java`"
- `↑,↓`. Repite comandos anteriores

Aunque existen muchos más comandos, el resto de acciones las realizaremos con entornos gráficos.

El primer programa

Utilizaremos el *Notepad++* para crear un fichero de texto. Se debe llamar *HolaMundo.java*. Siempre que una clase sea pública, el fichero se debe llamar igual que la clase. Cuidado con las mayúsculas y minúsculas, son diferentes.

A continuación presentamos el contenido del fichero:

```
public class HolaMundo {
   public static void main(String[] args) {
      System.out.println("---- Hola Mundo ----");
   }
}
```

El siguiente paso es compilar el fichero *java* y obtener un fichero *class*. Para ello, y desde la consola, nos situaremos en el directorio que contiene el fichero *java*. Sería conveniente ejecutar el comando *dir* y observar que aparece nuestro fichero java *HolaMundo.java*. A continuación ejecutaremos en la consola:

```
javac HolaMundo.java
```

Si no se producen errores de compilación, se habrá creado un fichero *HolaMundo.class*. Ejecutar el comando *dir* para comprobarlo.

Por último debemos ejecutar el fichero *class*, para ello utilizaremos el comando siguiente:

```
java HolaMundo
```

Al ejecutarse se imprimirá por la consola el mensaje "--- Hola Mundo ----".

Si entrar en muchos detalles, fijarse que hemos realizado una clase llamada *HolaMundo*. Esta clase tiene un sólo método (similar a procedimiento, función...) llamado *main*. Este método es importante porque es el único ejecutable desde la consola. La finalidad de nuestro método es imprimir un mensaje.

Ficharos JAR

Los ficheros *Jar* (*Java ARchives*) permiten recopilar en un sólo fichero varios ficheros diferentes, almacenándolos en un formato comprimido para que ocupen menos espacio. Además, para utilizar todas

las clases que contiene el fichero *Jar* no hace falta descomprimirlo, sólo se debe referenciar mediante la variable de entorno *CLASPATH*.

El fichero *Jar* puede ser ejecutable, para ello se debe añadir un fichero manifiesto donde indicamos la clase principal; es un fichero de texto con extensión "*mf*" (un ejemplo sería *manifest.mf*) e introducimos una línea con el contenido: `Main-class: MiClasePrincipal`.

Atención, se debe introducir un fin de línea (enter) para que funcione correctamente.

```
Main-Class: MiClasePrincipal

```

Recordar situar la ruta de la consola en la carpeta de trabajo y realizar un `dir` para asegurarse de la existencia de los ficheros. Para crear el *Jar*, utilizamos el comando *jar*:

```
jar cvfm ejecutable.jar manifest.mf Uno.class Dos.class ...
```

A partir de aquí, podemos ejecutar el fichero *ejecutable.jar* como si fuera una aplicación normal, realizando un doble clic desde un entorno gráfico. Si la aplicación Java no trabaja con ventanas, que sería el caso de nuestro ejemplo *HolaMundo* anterior, se debe ejecutar desde consola:

```
java -jar ejecutable.jar
```

Si queremos añadir los ficheros fuente de la aplicación, normalmente se crea un fichero comprimido donde se incorpora la carpeta *src* con los fuentes y el fichero *jar*.

Clase IO

La clase IO ha sido creada para facilitar la Entrada/Salida de datos en modo gráfico. Es una clase desarrollada propia y es independiente de la distribución Java SE. Se puede localizar en la ruta:

http://www.eui.upm.es/~jbernal/io.jar

En la actualidad, va por la versión 5.3.

Para su uso, se debe referenciar en la variable de entorno *classpath*. Además, se debe añadir la sentencia *import* en la clase:

```
import upm.jbb.IO;
public class HolaMundo {
...
```

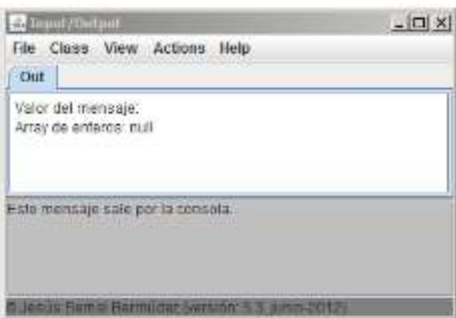


Figura 3. Clase IO

Los métodos más importantes son:

- Para leer un String: *IO.in.readString()* o *IO.in.readString("Mensaje")*
- Para leer un int: *IO.in.readInt()* o *IO.in.readInt("Mensaje")*
- Para leer un double: *IO.in.readDouble()* o *IO.in.readDouble("Mensaje")*
- Para leer cualquier clase: *IO.in.read("paquete.Clase", "Mensaje")*
- Para escribir un dato con salto de línea: *IO.out.println(...)*
- Para escribir un dato sin salto de línea: *IO.out.print(...)*
- Para escribir en la barra de estado: *IO.out.setStatusBar("mensaje")*

Para más información, en el menú *Help*, pulsar *Help Contents*

Un ejemplo de utilización de la clase IO es:

PruebaIO.java

```
import upm.jbb.IO;
public class PruebaIO {
   public static void main(String[] args) {
         String msg = IO.in.readString("Escribe un mensaje");
         IO.out.println("Valor del mensaje: " + msg);
         int entero = IO.in.readInt("Dame un número entero");
         double decimal = IO.in.readDouble("Dame un numero decimal");
         short corto = (short) IO.in.read("short", "Corto");
         Integer entero2 = (Integer) IO.in.read(8, "Valor por defecto");
         Byte b = (Byte) IO.in.read(new Byte("3"), "Valor por defecto");
         Integer[] intArray = (Integer[]) IO.in.read("Integer[]", "Array de enteros");
         IO.out.print("Array de enteros: ");
         IO.out.println(intArray);
         java.util.List<Byte> cb =
            (List<Byte>) IO.in.read("java.util.List<Byte>", "Colección");
         java.awt.Color color = (Color) IO.in.read("java.awt.Color", "color");
         IO.out.setStatusBar("Barra de estado");
         System.out.println("Este mensaje sale por la consola...");
   }
}
```

Podemos rescribir nuestro ejemplo de *HolaMundo* con la clase IO, el resultado es:

HolaMundo.java

```
import upm.jbb.IO;
public class HolaMundo {
   public static void main(String[] args) {
      IO.out.println("---- Hola Mundo ----");
   }
}
```

Para generar el fichero ejecutable, definiremos un fichero manifiesto. Se ha añadido una segunda línea para poder referenciar la librería *io.jar*:

manifest.mf

```
Main-Class: HolaMundo
Class-Path: ./io.jar
```

Recordar insertar un salto de línea al final de HolaMundo.

Y ejecutaríamos el comando jar:

```
jar cvfm hola.jar manifest.mf HolaMundo.class
```

Ahora, podremos realizar un doble clic a la aplicación *hola.jar* desde un entorno gráfico para ejecutarla, debe estar la librería io.jar en la misma carpeta.

Lenguaje de programación Java

Aspectos generales de la gramática

COMENTARIOS E IDENTIFICADORES

Los comentarios del código tienen tres formatos:

- Comentarios de línea: "//", es comentario hasta final de línea
- Comentarios de bloque: "/*" ... "*/"

- Comentarios de documentación: “/**” ...”*/”. Se utiliza para documentar los aspectos públicos de las clases; mediante la herramienta *javadoc.exe* se genera documentación en formato HTML con el contenido.

Los blancos, tabuladores y saltos de línea, no afectan al código.

Las mayúsculas y minúsculas son diferentes.

Los identificadores se forman mediante: {letra} + [letras | números | ”_” | “$”].

Existe un estilo de nombrar los diferentes elementos ampliamente aceptado por la comunidad Java:

- **Clases**: las palabras empiezan por mayúsculas y el resto en minúsculas (ej. HolaMundo)
- **Métodos**: las palabras empiezan por mayúsculas y el resto en minúsculas, excepto la primera palabra que empieza por minúsculas (ej. holaMundo)
- **Constantes**: todo en mayúsculas separando las palabras con “_”

DECLARACIÓN DE VARIABLES Y CONSTANTES

Existen dos tipos de variables: las ***primitivas*** y las de ***clase***. Las variables primitivas almacenan el dato en sí, por ello, dependiendo del tipo de dato ocupan un tamaño diferente de memoria. Las variables de clase almacenan la referencia del objeto y tienen un tamaño fijo de 32 bits.

La declaración de variables tiene el siguiente formato:

```
tipo identificador;
tipo identificador = expresión;
tipo identificador1, identificador2 = expresión;
```

Algunos ejemplos serían:

```
int prueba;
byte pruebaDos, prueba3 = 10, prueba4;
double prueba5 = 10.0;
```

Las constantes se declaran igual que las variables, pero añadiendo la palabra *final*.

```
final tipo IDENTIFICADOR = expresión;
```

Algunos ejemplos serían:

```
final int CONSTANTE_UNO = 10;
final double CONSTANTE_DOS = 10.0;
```

A continuación presentamos una tabla con los tipos primitivos, el tamaño que ocupa en memoria y el rango de valores que pueden representar.

- *Enteros*
  - ***byte***. 8 bits. -128 a +127
  - ***short***. 16 bits. -32.768 a +32.767
  - ***int***. 32 bits. -2.147.483.648 a -2.147.483.648
  - ***long***. 64 bits. -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
- *Decimales*
  - ***float***. 32 bits. $\approx \pm 1.7*10^{38}$
  - ***double***. 64 bits. $\approx \pm 3.4*10^{308}$
- *Caracteres*
  - ***char***. 16. ‘\u0000 ‘ a ‘\uffff’
- *Lógico*
  - ***boolean***. true y false
- *Objetos*
  - ***Object***

Vamos a realizar un breve adelanto de la clase String. Su utilidad es almacenar una cadena de caracteres. Un ejemplo de uso sería:

```
String nombre="Jorge", ciudad="Madrid";
```

Una cadena de caracteres no puede ser dividida mediante un salto de línea. Para cadenas largas, se dividen en varias y se concatenan con el operador “+”, pudiéndose definir en varias líneas.

Palabras reservadas

A continuación, se enumeran las palabras reservadas del lenguaje:

abstract, assert, Boolean, break, byte, case, catch, char, class, const, continue, default, do, double, else, enum, extends, final, finally, float, for, goto, if, implements, import, instanceof, int, interface, long, native, new, package, private, protected, public, return, short, static, strictfp, super, synchronized, switch, this, throw, throws, transient, try, void, volatile, while.

Literales

Los valores numéricos están en notación decimal. Para realizarlos en notación hexadecimal se añade “0x”, por ejemplo 0xffff. El tamaño por defecto de un entero es *int*, y de un decimal *double*. Para cambiar el tamaño a *long* se añade “L” al final (ej. 23L), para cambiar a *float* se añade ‘F”.

Los caracteres van entre comillas simples (ej. ‘x’). Los caracteres escape van precedidos por ‘\’, algunos ejemplos son:

- ‘\n’: retorno de carro
- ‘\t’: tabulador
- ‘\’’: comillas simples
- ‘\”’: comillas dobles
- ‘\\’: propia barra
- ‘\b’: pitido
- ‘\u----’: valor Unicode

Los caracteres Unicode van precedidos por el carácter ‘u’ y en notación hexadecimal, algunos ejemplos serían: ‘\u0064’, ‘\u1234’.

Operadores

Los operadores de Java son:

- *Asignación*
  - `=`. Ej.: `x=y;`
- *Aritméticos*
  - `++`, `--`. Suma o resta 1, ej.: `x++; ++x;`
  - `-` . Cambio de signo, ej.: `-x;`
  - `*`, `/`. Multiplicación y división, ej.: `x*y;`
  - `%`. Resto de la división, ej.: `x%y;`
  - `+`, `-`. Suma y resta, ej.: `x+2;`
  - `+=`, `-=`, `*=`, `/=`, `%=`. Ej.: `x+=2; x=x+2;`
- *Binarios*
  - `~`. Not, ej.: `~x;`
  - `<<`. Desplazamiento a la izquierda, se introducen ceros, ej.: `x<<2;`
  - `>>`. Desplazamiento a la derecha, se mantiene signo, ej.: `x>>2;`
  - `>>>`. Desplazamiento a la derecha introduciendo ceros
  - `&`, `|` y `^`. And, or y xor, ej.: `x&12;`

- Comparación
  - `==`, `¡=`, `<`, `>`, `<=`, `>=`. Igual, distinto, menor que...
- Lógicos
  - `!`. Not lógico
  - `&`, `&&`, `|`, `||` y `^`. And, and perezoso, or, or perezoso y xor

El operador "+" entre String realiza la función de concatenación.

Expresiones

Las expresiones son una combinación de operadores y operandos. La precedencia y asociatividad determina de forma clara la evaluación. El orden aplicado es:

- Paréntesis: ( )
- Unarios: + - !
- Multiplicativos: * / %
- Aditivos: + -
- Relacionales: < > <= >=
- Igualdad: == !=
- Y lógico: & &&
- O lógico: | ||

Arrays

Los arrays son tablas de valores del mismo tipo. En Java los arrays son dinámicos y deben crearse o reservar memoria en tiempo de ejecución para su utilización. Por lo tanto, para poder utilizar arrays primero se debe declarar, y posteriormente, reservar la memoria para su uso.

Para definir un array tiene el siguiente formato:

```
tipo[] identificador;
```

Se define pero no se crea, realmente el contenido de una variable de array es una referencia a una zona de memoria; el valor inicial es la palabra reservada *null*. Un ejemplo sería:

```
int[] diasMes;
```

Para reservar memoria para el array se debe indicar el tamaño; una vez establecido el tamaño, los posibles valores de índice van desde 0 a tamaño-1:

```
tipo[] identificador = new tipo[tamaño];
```

Si se vuelve a reservar memoria, se libera la memoria anteriormente utilizada y por lo tanto se pierde el contenido que tuviese.

Las dos acciones anteriores se pueden realizar en una sola línea:

```
int[] tabla = new int[10];
```

También se puede declarar un array, reservar memoria y darle contenido en una sola línea:

```
int[] tabla = { 0, 1, 2 };
```

Para saber el tamaño de un array no nulo (ya se ha reservado espacio en memoria) se puede utilizar el atributo *length*: `identificador.length`. Si se intenta consultar el tamaño de un array no creado da un error (*NullPointerException*).

Se pueden definir arrays de varias dimensiones. Por ejemplo:

```
int[][] matriz = new int[10][20];
```

Para acceder a una posición del array se realiza a través de su índice, tienen un rango fijo de 0 a tamaño-1. Algunos ejemplos son:

```
int i = tabla[2];
i = matriz[2][3];
```

Si una variable de tipo array lleva el calificador *final*, quiere decir que el array no se permite que se vuelva a definir con el calificador *new*, pero si se permite cambiar el contenido del array. Mostramos un ejemplo para comprender este concepto:

```
final int[] array = new int[5];
array[3] = 3; // Correcto
// Error con array = new int[10];
```

PROMOCIÓN Y CASTING

Cuando en Java operamos con expresiones, los resultados intermedios se almacenan en variables temporales con el tipo de mayor capacidad que intervenga en la operación; excepto con *byte* y *short* que se promociona automáticamente a *int*.

El *casting* es forzar la conversión de un tipo a otro. Esta conversión podría dar errores si son incompatibles. La forma de realizarlo es situando delante, y entre paréntesis, el nuevo tipo. Presentamos varios ejemplos:

```
int i = 3;
byte b = (byte) i;
byte b2 = 2;
b2 = (byte) (b * 3);
```

Hay que tener cuidado que en las operaciones parciales no se pierda precisión. En la siguiente expresión: `int/int + double`, el valor parcial de la división se guarda en tipo *int*, y por lo tanto se pierden los decimales.

Para solucionar este problema tenemos dos opciones: `(double)int/int + double` o `1.0*int/int + double`.

Recordar que los valores enteros son, por defecto, *int*; y los valores decimales *double*.

Sentencias

En general, todas las sentencias acaban con "`;`".

SENTENCIA DE BLOQUES

Agrupa en una sentencia de bloque un conjunto de sentencias. Para ello se sitúan entre llaves.

```
{ sentencia1; sentencia2; ...}
```

Sentencias condicionales

SENTENCIA IF

La sentencia *if* es una de las más básicas y nos permite controlar el flujo de control dependiendo de una condición. Los dos formatos posibles son:

```
if (condicion) sentencia1;
if (condición) sentencia1; else sentencia2;
```

Si en lugar de una sentencia tenemos varias, se debe utilizar la sentencia de bloque. Algunos ejemplos son:

```
if (a == 3) {
    b = 8;
}
if (a > 2) {
    a -= 1;
    b += 1;
```

```
    }
    if (a > 2) {
        a -= 1;
        b += 1;
    } else a += 1;
```

Todas las sentencias se pueden anidar. Recordar que se debe aplicar un tabulador para su mejor comprensión. Un ejemplo sería:

```
    if (a > 2) {
        if (b == 8) {
            b++;
        }
    } else {
        b--;
    }
```

SENTENCIA ELSE-IF

Aunque la sentencia *else-if* no existe como tal, queremos presentarla de forma independiente ya que debe tener un estilo peculiar de tabulación. Se considera una sentencia *else-if* si en la condición siempre se evalúa una misma variable o expresión; aunque existan anidamientos de sentencias, los tabuladores aplicados y el uso de llaves son diferentes a las normas generales. Un ejemplo sería:

```
    if (a==0) {
        sentencia1;
    } else if (a == 1 | a == 2) {
        sentencia2;
    } else if (a < 10) {
        sentencia3;
        sentencia4;
    } else {
     // No se debiera llegar aquí
    }
```

SENTENCIA SWITCH

Esta sentencia evalúa una expresión, y dependiendo de su valor, ejecuta un conjunto de sentencias. Sólo se puede aplicar a valores primitivos, enumerados y a clases especiales. Su formato es:

```
    switch (Expresion) {
        case valor:
            sentencia1;
            sentencia2;
            break;
        case valor2:
        case valor3:
            sentencia3;
            break;
       case valor4:
            sentencia4; break;
        default: sentencia5;
    }
```

Sentencias de bucles

SENTENCIA WHILE Y DO WHILE

Estas sentencias se utilizan para realizar bucles o repeticiones hasta que se cumpla una determinada condición, pero a priori es indefinido el número de veces que se puede repetir.

La sentencia *while* tiene una repetición de 0 a n, y la sentencia *do while* de 1 a n. Su formato es:

```
    while (condición) {
        sentencia;
        sentencia;
        sentencia;
    }
```

```
do {
    senten2;
} while (condición)
```

SENTENCIA FOR

Esta sentencia realiza un bucle, donde existe un índice de iteración y la condición de fin depende del índice. Tiene tres partes separadas por "`;`": la primera se inicializan los índices (si hay más de uno se separan por "`,`"), la segunda es la condición de terminación (el bucle se repite mientras se cumpla la condición) y la tercera es el incremento de los índices.

Si los índices se declaran en el propio *for*, no se podrán utilizar fuera de éste. Normalmente, la condición debe depender del valor del índice.

Su formato es:

```
for (inicialización; terminación(mientras); operación) {
   sentencia1;
  ...
}
```

Mostramos algunos ejemplos:

```
for (int i = 1; i <= 10; i++) {
    a *= 2;
}

int i, j;
for (i = 1, j = 1; (i <= 5) && (j <= 5); i++, j += 2) {
    // ...
}

for (;;) {
    // ...
} // bucle infinito
```

SENTENCIA CONTINUE Y BREAK

Con la sentencia *continue*, se continúa con la siguiente interacción del bucle; puede aparecer el varios lugares del bucle, pero normalmente asociado a una condición.

Con la sentencia *break* abortamos el bucle, o en general, el bloque donde se ejecute.

Es posible que necesitemos un bucle con índice y además una condición especial de finalización, en este caso, se recomienda el bucle *for* con la sentencia *break* para la condición especial.

Un ejemplo sería:

```
int a = 1;
for (int i = 1; i <= 100; i++) {
   if (a % 5 == 0) continue;
   if (a > 1000) break;
   a += a * 2 + 1;
}
```

SENTENCIA FOR EACH

Esta sentencia es útil para procesar una colección de datos, donde sólo nos interesa el conjunto de datos y no el orden con que se procesan.

Se puede aplicar a las clases *Collection* y a los *arrays*. Su formato es:

```
for (int item: array){
    // en la variable item tenemos el dato
}
```

Un posible ejemplo sería:

```java
int[] datos = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

for (int item : datos) {
    System.out.println(item);
}
```

Es equivalente al siguiente bucle:

```java
for (int i=0; i<datos.length;i++) {
    System.out.println(datos[i]);
}
```

✍ ***Realizar la clase BucleBasicoTriple. A partir de un valor leído mediante IO (le llamaremos n), escribir la secuencia 0, 1, 2... hasta n tres veces, la primera realizada con la estructura while, la segunda con do...while y la tercera con for.***

✍ ***Realizar la clase Serie1. Dado un valor leído mediante IO (n), calcular el sumatorio: n + (n-1) + (n-2)+...+1; por ejemplo, para n=4 sería 4+3+2+1 = 10.***

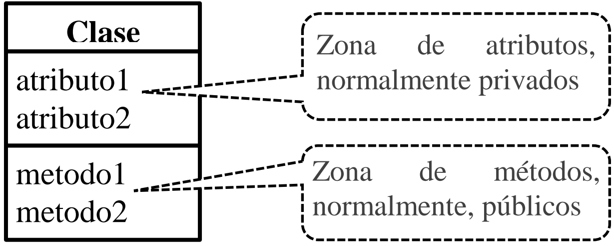
✍ ***Realizar la clase Tabla. Se define una array con el contenido {3,2,6,8,4}, calcular la suma de todos los elementos dos veces, una mediante la estructura for y otra mediante for each.***

✍ ***Realizar la clase Tabla2. Se define una array con el contenido {3,2,6,8,4}, ordenar el array de menor a mayor.***

Clases y objetos

Clases: atributos y métodos

Como ya definimos anteriormente, una clase es un conjunto de atributos y métodos. Vemos en la figura siguiente, una manera de mostrar una clase.



Las clases tiene dos vistas: una pública y una privada. La vista pública, también llamado interfaz, establece los aspectos visibles de la clase, y por lo tanto, nos ofrece los métodos que podemos utilizar. La vista privada, también llamada implantación, encapsula todos los detalles de la estructura de datos y la implementación de los métodos.

Atributos

Normalmente, los atributos son privados, y por lo tanto, no son directamente accesibles. La consulta o modificación de los atributos se debe realizar a través de métodos públicos; de esta manera, el objeto controla las operaciones válidas que se pueden realizar.

Un atributo es una información que se almacena y es parte representativa del estado. Los atributos se pueden sustentar mediante tipos primitivos o clases. Mostremos algunos ejemplos.

La clase *Punto* representa un punto en un espacio de dos dimensiones, sus atributos son *x* e *y*. La declaración de la clase quedaría:

```
public class Punto {
    private int x, y;
}
```

Observar, que los atributos se declaran a nivel de instancia, fuera de los métodos, y por lo tanto, son accesibles de cualquier parte de la clase.

La clase *NumeroNatural* representa un valor natural, su atributo podría ser *valor*. La clase quedaría:

```
public class NumeroNatural{
    private int valor;
}
```

La clase Usuario representa a un usuario, podría tener los atributos *nombre*, *eMail*, *movil*, *ciudad* y *mayorEdad*. El código java sería:

```
public class Usuario{
    private String nombre, eMail, ciudad;
    private long movil;
    private boolean mayorEdad;
}
```

Como puede observarse en todos los ejemplos, los atributos se declaran a nivel de la clase y están calificados con *private*, indicando que no es accesible desde fuera de la clase.

Se pueden crear tantos objetos o instancia de las clases como queramos. Dos objetos de la clase *Punto* tienen valores de atributos independientes y evolucionan por separado; eso sí, cuando se crea la instancia, todas parten del mismo estado.

Métodos

La razón de establecer los atributos en vista privada, es poder controlar la modificación del estado del objeto y no permitir que este evolucione hacía estados incoherentes. Pero debemos establecer los mecanismos para poder modificar el objeto, y son los métodos, el sistema adecuado.

Podemos organizar los métodos en tres tipos, teniendo en cuenta aspectos sintácticos y semánticos:

- Constructores. Métodos que inicializan la instancia
- Métodos genéricos. Realizan acciones utilizando los atributos
- Métodos para accesos directo a los atributos (*getters & setters*). Estos métodos son opcionales y suelen utilizarse más en clases de tipo “entidad”, es decir, que tienen persistencia en bases de datos, o en los “beans”, donde tienen una correspondencia con formularios. Aunque estos métodos son realmente generales, los distinguimos por separado por tener unas normas concretas de estilo.

MÉTODOS GENÉRICOS

Comencemos con el formato básico de un método:

```
[public | private] {void|tipo} identificadorMetodo (tipo param1, tipo param2...) {...}
```

El primer calificador es *public* o *private*, se establece el tipo de vista.

La segunda palabra es el tipo devuelto, con el calificador *void* (vacío) se indica que no devuelve nada, o se establecer el tipo devuelto.

La siguiente palabra representa el nombre del método.

A continuación vienen los parámetros. Hay que resaltar, en general, los datos con que actúan los métodos son sus propios atributos, y por lo tanto no es necesario pasarlos por parámetro. Por parámetros se pasan valores externos al objeto que condicionan la acción a realizar.

Existen dos tipos de parámetros:

- Tipos primitivos (byte, short, int...). Estos se pasan por *valor*, es decir, se realiza una copia del contenido en la variable que sustenta el parámetro, y por lo tanto, si el método modifica el parámetro no afecta a la variable del mensaje entrante.
- Clases y arrays. Estos se pasan por *referencia*, es decir, se pasa la dirección en el parámetro, así que, modificaciones dentro del método afectan externamente.

Con la sentencia *return*, se finaliza el método. Si además queremos devolver un valor se utiliza la sentencia *return valor*.

Ampliando la clase Punto, se podrían añadir los siguientes métodos:

```
public class Punto{
   private int x, y;
   public double modulo() {
      double valor = 0;
      // Se calcula el modulo del vector con x,y
      return valor;
   }
   public double fase() {
      double valor = 0;
      // Se calcula la fase del vector con x,y
      return valor;
   }
}
```

Dentro de los métodos podemos definir variables locales, las cuales serán creadas en la ejecución del método y destruidas al finalizar la ejecución. Las variables de método, sólo son visibles desde el propio método en que se definen.

MÉTODOS PARA ACCESOS A ATRIBUTOS (GETTERS & SETTERS)

Realmente, a nivel gramatical, se pueden considerar como métodos genéricos. Pero semánticamente, tienen connotaciones especiales que debemos especificar.

Cada atributo tiene dos posibles acciones: leer su valor o establecerlo. No tenemos por qué realizar ambas acciones, depende del diseño que se busque, de hecho, muchas clases no tiene estos métodos.

Existe una nomenclatura especial para el identificador del método:

- Establecer el valor de un atributo: *set* + *NombreAtributo* (la primera letra del nombre del atributo debe estar en mayúsculas). Por ejemplo, en la clase *Punto* sería *setX* y *setY*, en la clase *NumeroNatural* sería *setValor*, en la clase *Usuario* sería *setNombre*, *setEMail*, *setCiudad*, *setMovil* y *setMayorEdad*.
- Leer el valor del atributo: *get* + *NombreAtributo* o *is* + *NombreAtributo* si devuelve un tipo *boolean*. Por ejemplo, en la clase Punto sería *getX* y *getY*, en la clase *NumeroNatural* sería *getValor*, en la clase Usuario sería *getNombre*, *getEMail*, *getCiudad*, *getMovil* y *isMayorEdad*.

A continuación, presentamos la estructura completa del código de la clase Punto:

Punto.java

```
public class Punto{
   private int x,y;
   public void setX(int x){
      //...
   }
   public void setY(int y){
      //...
   }
   public int getX(){
      return x;
   }
   public int getY(){
      return y;
   }
   public double modulo(){
      //...
   }
   public double fase(){
      //...
```

```
    }
}
```

CONSTRUCTORES

Los constructores son métodos especiales que reúnen las tareas de inicialización de los objetos de una clase; por lo tanto, el constructor establece el estado inicial de todos los objetos que se instancian.

No es obligatorio definir constructores, si no se realiza, existe un constructor por defecto sin parámetros.

La ejecución del constructor es implícita a la creación de una instancia.

La restricción sintáctica del constructor es que debe llamarse igual que la clase y no devuelve ningún tipo ni lleva la palabra *void*.

Como ejemplo añadimos un constructor a la clase Punto:

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

SOBRECARGA DE MÉTODOS

La sobrecarga es definir dos o más métodos con el mismo nombre, pero con parámetros diferentes por cantidad o tipo. El objetivo de la sobrecarga es reducir el número de identificadores distintos para una misma acción pero con matices que la diferencian. La sobrecarga se puede realizar tanto en métodos generales, como en constructores.

La sobrecarga es un polimorfismo estático, ya que es el compilador quien resuelve el conflicto del método a referenciar.

Si definimos un constructor con parámetros, el constructor sin parámetros deja de estar disponible; así que, si nos interesa, se debe definir para su utilización.

Como ejemplo añadimos sobrecarga a los constructores de la clase Punto:

```
public class Punto{
   private int x,y;
   public Punto(int x, int y){}
   public Punto(int xy){}
   public Punto(){}
   ...
}
```

Objetos

Recordemos que la clase representa el modelo para crear objetos, pero son los objetos los elementos con los que podemos trabajar. Los objetos ocupan memoria y tiene un estado, que representa el conjunto de valores de sus atributos.

Podemos crear todos los objetos que queramos, todos ellos tiene existencia propia, pudiendo evolucionar por caminos diferentes e independientes.

Cuando se crea un objeto se debe asociar con una referencia, por lo tanto, el proceso debe tener dos partes:

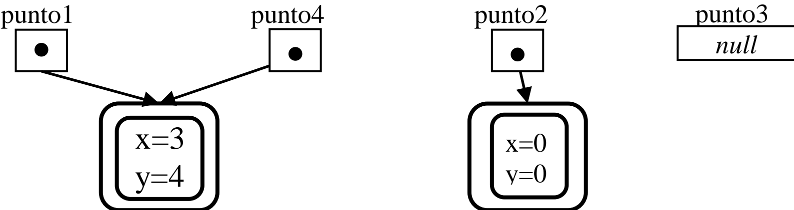
1. Declarar una variable que pueda referenciar al objeto en cuestión
2. Crear el objeto y asociarlo a la variable. Para crear el objeto, se realiza con la sentencia ***new***, recordar que lleva implícito llamar al método constructor para que inicialice el objeto

Un ejemplo sería:

```
      Punto punto1, punto2 = new Punto(), punto3, punto4;
      punto1 = new Punto(3,4);
```

```
punto4 = punto1;
```

En este ejemplo declaramos cuatro puntos de la clase Punto. Cada vez que se utiliza la palabra ***new***, quiere decir que se crea una nueva instancia, por lo tanto, tenemos un total de dos instancias. En el ejemplo se han utilizado dos constructores diferentes para inicializar las instancias. Mostramos a continuación, una gráfica una vez ejecutado todo el código.



Fijarse que las variables *punto1* y *punto4* referencian la misma instancia, es una manera redundante de acceder. El contenido de una variable de instancia es la dirección de memoria donde se encuentra el objeto. Cuando comparamos variables de instancia, realmente comparamos direcciones de memoria.

Se ha declarado la variable *punto3*, pero no se ha asociado a ninguna instancia, en ese caso, su contenido es *null*. Podemos asignar a cualquier variable de instancia el valor *null*, por ejemplo `punto2=null`.

En Java, no existe ninguna forma explícita para destruir un objeto, simplemente cuando un objeto se queda huérfano (no tiene ninguna referencia) se destruye. Esta labor la realiza el recolector de basura (*garbage collector*) de forma automática. En el ejemplo anterior, si asignamos a *punto2* un *null*, queda su instancia sin referencia, por consiguiente, cuando se lance el recolector de basura destruirá el objeto huérfano.

Notar que cuando creamos un array NO implica la creación de los objetos. Veamos un ejemplo en tres pasos para entender mejor el proceso.

- En primer lugar se declara una variable de tipo array de Punto.

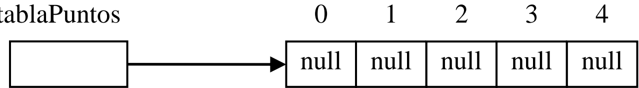
```
Punto[] tablaPuntos;
```

El contenido de la variable *tablaPuntos* es *null*.

- En segundo lugar se crea el array.

```
tablaPuntos = new Punto[5];
```

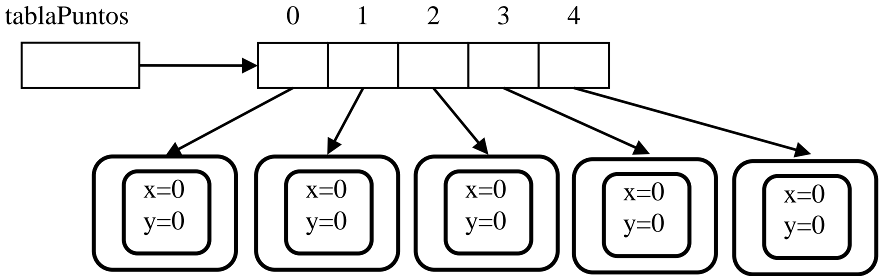
Gráficamente se describiría como:



- En tercer lugar se crean los objetos.

```
for (int i = 0; i < 5; i++)
    tablaPuntos[i] = new Punto();
tablaPuntos[i]= new Punto();
```

Gráficamente se describiría como:



Si una variable de Clase lleva el calificador *final*, quiere decir que no se permite que se vuelva a crear una nueva instancia con el calificador *new*, pero si se permite cambiar el contenido de la instancia. Mostramos un ejemplo para comprender este concepto:

```
final Punto pto = new Punto();
pto.setPtX(3); //Correcto
//Error con pto = new Punto();
```

Mensajes

Una vez creado los objetos es posible pasarles mensajes, o invocarles métodos, para solicitar una acción. El objeto establece a que método le corresponde el mensaje recibido.

Para enviar un mensaje se realiza con la notación *punto*. Dependiendo si almacenamos el valor devuelto del método, tenemos dos posibilidades:

```
referencia.metodo(param1,param2...);
referencia.metodo();

var = referencia.metodo(param1,param2...);
var = referencia.metodo();
```

La segunda línea es para guardar el valor devuelto por el mensaje.

Ponemos a continuación un ejemplo con la clase Punto:

```
Punto punto1 = new Punto(2,5);
punto1.setX(4);
int x = punto1.getX();
double d = punto1.modulo();
punto1.setX(punto1.getX() + 1); // suma en una unidad el valor de x
```

THIS

Dentro de la implementación de un método, a veces se necesita referenciar a la propia instancia a la cual pertenece. Para ello está la palabra reservada *this*. Un ejemplo dentro de *Punto* sería: `this.x = 3`.

Resulta recomendable calificar con *this* a todas las referencias a los atributos y métodos de la propia clase; y resulta obligatorio cuando queramos distinguir entre un parámetro y una variable de instancia con el mismo nombre.

También referenciamos con la palabra *this* a nuestros propios constructores, por ejemplo `this(3,4)`.

A continuación presentamos el código completo de la clase Punto:

Punto.java

```
public class Punto {
   private int x, y;
   public Punto(int x, int y) {
      this.setX(x);
      this.setY(y);
   }
   public Punto(int xy) {
      this(xy, xy);
   }
   public Punto() {
      this(0, 0);
```

```
    }
    public void setX(int x) {
        this.x = x;
    }
    public void setY(int y) {
        this.y = y;
    }
    public int getX() {
        return this.x;
    }
    public int getY() {
        return this.y;
    }
    public double modulo() {
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }
    public double fase() {
        double aux = (double) (this.y) / this.x;
        return Math.atan(aux);
    }
}
```

Se debe evitar la redundancia de código, para ello sólo se implementa un solo constructor y el resto referencian a este, mediante valores por defecto de los parámetros. Los constructores deben hacer uso de los métodos para el acceso a los atributos, y así, se aplica los filtros o controles pertinentes.

Otro ejemplo es la clase *NumeroNatural*:

NumeroNatural.java

```
public class NumeroNatural {
    private int natural;
    public NumeroNatural(int natural) {
        this.setNatural(natural);
    }
    public NumeroNatural() {
        this(0);
    }
    public int getNatural() {
        return this.natural;
    }
    public void setNatural(int natural) {
        if (natural < 0) this. natural = 0;
        else this.natural = natural;
    }
    public void añadir(int valor) {
        //this.natural += valor; //Error se salta el filtro
        this.setNatural(this.natural + valor);
    }
}
```

✍ *Realizar la clase Rango y PruebaRango, representa un número natural limitado hasta un máximo.*

- Atributos:
  - valor (lectura y escritura). En la escritura se deben filtrar lo valores fuera de rango
  - maximo (sólo lectura). Sólo lectura significa que el método *setMaximo* está calificado con *private* o, simplemente, no se define el método.
- Constructores:
  - Rango(int maximo, int valor)
  - Rango()
- Métodos:
  - void reset(), pone *valor* a cero

✍ ***Realizar la clase ContadorCircular y PruebaContadorCircular, representa un contador similar a un cuentakilómetros de un coche, con cada petición incrementa en uno el contador y al llegar al valor máximo vuelve a cero.***

- Atributos:
  - maximo (sólo lectura) representa el valor máximo de cuenta, sólo se puede establecer con el constructor
  - contador (sólo lectura)
- Constructores:
  - ContadorCircular(int maximo)
- Métodos:
  - void reset(), pone a cero el contador
  - void incrementar(), incrementa en una unidad el contador. Cuando se pase del máximo pasa automáticamente a cero

Elementos estáticos

Todos los conceptos expuestos giran en torno a una idea principal: la clase describe el comportamiento de cada uno de sus objetos.

El presenta apartado pretende dar solución a cuestiones de la clase y de todos los objetos en su generalidad. Por ejemplo, si definimos una constante de una clase que tenga siempre el mismo valor, es común a todos los objetos, y por lo tanto, debe estar asociado a la clase y no a las instancias.

Podemos definir atributos estáticos y métodos estáticos.

ATRIBUTOS ESTÁTICOS

Para definirlos se antepone al tipo la palabra *static*; se deben inicializar independientemente a los constructores.

Estos atributos están compartidos por todos los objetos de la clase y son accesibles desde cada uno de ellos. Cualquier modificación por parte de un objeto afectará al resto.

Para el acceso a un atributo, en lugar de utilizar *this*, se utiliza el nombre de la clase: *Clase.atributo*.

Normalmente, las constantes asociadas a la clase son buenas candidatas para realizarlas estáticas.

Un ejemplo sería:

```
public class MiClase{
  private static final int MAX = 100;
  //...
}
```

MÉTODOS ESTÁTICOS

Al igual que los atributos, se antepone la palabra *static* al tipo devuelto.

No puede utilizar en el código la palabra *this*, ya que está asociado exclusivamente a las instancias. Se pueden acceder a los atributos estáticos del mismo modo que pueden las instancias, anteponiendo el nombre de la clase: *Clase.atributo*.

Se permite definir un código estático para inicializar los atributos estáticos. Su ejecución se dispara una sola vez al principio. Tiene el siguiente formato:

```
  static {
     //...
  }
```

Vamos añadir algunos aspectos estáticos a la clase Punto:

Punto.java
```java
public class Punto {
   public static final double PI = 3.1415;
   private static int numeroInstancias;
   private int x, y;
   static {
      Punto.numeroInstancias = 0;
   }
   //Contructores. Deben incrementar el atributo numeroInstancias
   public static int instanciasCreadas() {
      return Punto.numeroInstancias;
   }
   //...
}
```

PruebaPunto.java
```java
public class PruebaPunto {
   public static void main(String[] args){
      Punto p;
      p = new Punto(2,2);
      p = new Punto(2,2);
      p = new Punto(2,2);
      System.out.println(Punto.instanciasCreadas());
      System.out.println(Punto.PI);
   }
}
```

Clases de Java fundamentales

Clase String

La clase String representa una cadena de caracteres. Su contenido no puede cambiar, por lo tanto, cada vez que se modifica, se crea un nuevo String. Consume menos recursos para el almacenamiento pero las operaciones de modificación resultan costosas en el tiempo. Si van a existir numerosas modificaciones se deberá utilizar la clase StringBuffer.

Todos los literales de cadena como *"abc"* se implementan como una instancia de String, podemos realizar la siguiente asignación:

```java
String s = "abc";
```

Si se quiere crear cadena a partir de valores o variable nativas, debemos utilizar el método estático valueOf(...). Algunos ejemplos son:

```java
String s;
s = String.valueOf(3);
s = String.valueOf(3.3);
s = String.valueOf('a');
s = String.valueOf(true);
```

El operador "+" entre cadenas se interpreta como concatenación. De forma automática, al concatenar un String con cualquier otro valor, lo convierte a String. Algunos ejemplos son:

```java
String s = "abc", s2;
int i = 1;
s2 = s + i; // "abc1"
s2 = "" + 3.4; // "3.4"
```

La clase String tiene un gran número de métodos que nos facilita el trabajo con cadenas. Algunos de los más útiles son:

- Longitud de la cadena (*length*).
- Comparar cadenas (*equals*, *equalsIgnoreCase*, *compareTo*).
- Busca subcadenas (*indexOf*, *lastIndexOf*)
- Extrae subcadenas (*substring*, *toLowerCase*, *toUpperCase*, *trim*)

Algunos ejemplos de uso son:

```
String s = "abcde";
int l = s.length(); // 5
String s2 = s.toUpperCase(); // ABCDE
int i = s.indexOf("cd"); // 2
String s3 = s.substring(1, 3); // bc
```

CLASE STRINGBUFFER

Representa una cadena de caracteres con una estructura interna que facilita su modificación, ocupa más que la clase String, pero las operaciones de modificación son muchos más rápidas.

Para la modificación del contenido, cuenta con los métodos *insert* y *append*. Mirar el API para mayor documentación.

En el siguiente código, el bucle realizado con *StringBuffer* es aproximadamente 1000 veces más rápido que el bucle realizado con *String*. Es importante tener en cuenta, que cuando se requiera modificar una cadena de forma dinámica, es mucho mejor la clase *StringBuffer*.

```
String s = "";
StringBuffer sb = new StringBuffer();

for (int i = 0; i < 100000; i++) {
    s += ".";
}

for (int i = 0; i < 100000; i++) {
    sb.append(".");
}
```

CLASE NUMBER: INTEGER, DOUBLE...

Este conjunto de clases son útiles para trabajar con los valores primitivos, pero mediante objetos. A estas clases se les llama *Wrappers* (envolventes). Cada clase especializada, almacena un tipo de valor primitivo numérico.

Las clases tienen constantes que nos pueden ser útiles, por ejemplo, todas tienen MAX_VALUE y MIN_VALUE, *Double* además tiene NaN, POSITIVE_INFINITY. NEGATIVE_INFINITY...

Otra utilidad interesante es que pueden convertir una cadena a un valor numérico, por ejemplo, Integer.parseInt("123") devuelve el valor *int* asociado.

Para facilitar su uso, en las versiones actuales de Java existe una conversión automática entre el tipo primitivo y el objeto (*boxing* y *unboxing*). El siguiente código funciona correctamente:

```
Integer claseI;
Double claseD;
int i;
double d;
claseI = 23;
claseD = 2.2;
i = claseI;
d = claseD;
```

BOOLEAN, CHARACTER, MATH, SYSTEM...

Mirar el API para mayor documentación.

Javadoc

Los comentarios *JavaDoc* están destinados a describir, principalmente, las clases y los métodos. Existe una utilidad de Sun Microsystems llamada *javadoc.exe* para generar APIs en formato HTML de un documento de código fuente Java.

Recordar que los comentarios *javadoc* son /** ... */. Se utiliza el carácter @ como anotación especial, para describir las etiquetas de documentación.

Los comentarios se deben colocar justo antes del elemento a documentar.

Se permite añadir elementos HTML a la documentación para enriquecer su presentación.

ANOTACIONES

Generales

- **@see** *referencia*. Realiza un enlace. Ejemplos de referencia son: #metodo(); clase#metodo(); paquete.clase; paquete.clase#metodo(); enlace estándar de html
- **@since**. Indica en la versión que apareció este elemento respecto a la secuencia de versiones aparecidas a lo largo del tiempo

Paquetes

- **@see**, **@since**
- **@deprecated**. Indica que este elemento es antiguo y que no se recomienda su uso porque posiblemente desaparecerá en versiones posteriores. Se debe indicar la alternativa correcta

Clases e interfaces

- **@see**, **@since**, **@deprecated**
- **@version**. Versión y fecha
- **@author**. Nombre del autor

Variables y constantes

- **@see**, **@deprecated**

Métodos

- **@see**, **@since**, **@deprecated**
- **@param**. Definición de un parámetro, es obligatorio. *@param *parámetro* descripción
- **@return**. documenta el dato devuelto, es obligatorio. *@return descripción
- **@throws**. Excepción lanzada por el método. *@throws clase descripción

Un ejemplo de uso podría ser:

Mes.java

```
/**
 * La clase Mes representa un mes. Esta clase puede trabajar con valores
 * enteros: 1..12, o con strings: "enero".."diciembre".
 *
 * @author Jesús Bernal Bermúdez
 * @version 1.0 04/03/2008
 * @since 1.0
 */
public class Mes {
   /**
    * Array con los strings de los doce meses
    */
   public static final String[] MESES = { "enero", "febrero", "marzo", "abril", "mayo",
         "junio", "julio", "agosto","septiembre", "octubre", "noviembre", "diciembre" };
   private int mes; // Tipo primitivo que almacena el mes de la instancia.

   /**
    * Crea una nueva instancia a partir de un valor entero.
    *
    * @param mes
    *            int que representa el mes.
    */
   public Mes(int mes) {...}

   /**
    * Crea una nueva instancia y se inicializa con el mes 1, "enero".
    */
   public Mes() {...}
```

```java
/**
 * Crea una nueva instancia a partir de un string. Si no se reconoce el
 * string se interpreta "enero"
 *
 * @param mes
 *           String que representa el mes. No se distinguen mayúsculas de
 *           minúsculas.
 */
public Mes(String mes) {...}

/**
 * Establece un nuevo mes.
 *
 * @param mes
 *           int que representa el mes.
 */
public void setMes(int mes) {...}

/**
 * Establece un nuevo mes a partir de un string. Si no se reconoce el string
 * se interpreta "enero"
 *
 * @param mes
 *           String que representa el mes. No se distinguen mayúsculas de
 *           minúsculas.
 */
public void setMes(String mes) {...}

/**
 * Devuelve el mes que representa la instancia en formato numérico.
 *
 * @return int que representa el mes.
 * @see #getMes()
 */
public int getMes() {...}

/**
 * Devuelve el mes que representa la instancia en formato string.
 *
 * @return String que representa el mes.
 */
public String getMesString() {...}

/**
 * Se pasa al mes siguiente.
 */
public void incrementa() {...}

/**
 * Devuelve un String que reprersenta el objeto.
 *
 * @return un String que representa al objeto
 */
public String toString() {...}

/**
 * Devuelve una nueva instancia que es una copia de la actual.
 *
 * @return Mes que representa un clon de la actual
 */
public Mes clonar() {...}

/**
 * Devuelve el string correspondiente a un entero. Si el valor entero no es
 * correcto se devuelve null.
 *
 * @param valor
 *           int que representa el mes a convertir.
 * @return String asociado al mes.
 */
public static String literalDe(int valor) {...}

/**
 * Devuelve el entero correspondiente a un mes.No se distinguen mayúsculas de
 * minúsculas. Si no se reconoce el String se devuelve 0.
 *
 * @param valor
 *           String que representa el mes a convertir.
```

```
     * @return int asociado al mes.
     */
    public static int valorDe(String valor) {...}
}
```

Para generar la documentación en *html*, se realiza de forma parecida a compilar, desde la consola se teclea: `javadoc Mes.java`. Conviene realizarlo en una carpeta independiente debido a la gran cantidad de ficheros que genera. Si queremos realizarlos con varias clases sería:

```
javadoc C1.java C2.java C3.java
```

```
javadoc *.java
```

✍ Implementa por completo la clase Mes

✍ Realizar la clase Pila, gestiona una pila de cadenas (String) (último en entrar, primero en salir). Para su implementación se apoya en un array.

- Atributos:
  - capacidad (sólo lectura) representa el tamaño del array. Sólo se puede establecer en el constructor
  - array (sin acceso). Significa que los métodos get/set son *private* o no se implementan
  - cima (sin acceso) apunta al último elemento apilado, -1 si no hay elementos
- Constructores:
  - Pila(int capacidad), Pila()
- Métodos:
  - void borrarTodo(), borra toda la pila
  - boolean apilar(String elemento). Añade un elemento en la cima de la pila, si no está llena. Devuelve un valor lógico indicando si se ha podido añadir
  - String desapilar(). Devuelve el String que se encuentra en la cima de la pila, devuelve null si la pila está vacía
  - int tamaño(). Devuelve el número de String que contiene la pila

La clase *TestPila* debe tener el siguiente contenido:

TestPila.java

```
import upm.jbb.IO;

public class PilaTest {
    private Pila pila;

    public PilaTest() {
        pila = new Pila(3);
        IO.in.addController(this);
    }

    public void borrarTodo() {
        pila.borrarTodo();
        IO.out.println(">>>" + pila.toString());
    }

    public void apilar() {
        IO.out.println(">>> " + pila.apilar(IO.in.readString("Apilar")) +
              ">>>" + pila.toString());
    }

    public void desapilar() {
        IO.out.println(">>> " + pila.desapilar() + ">>>" + pila.toString());
    }

    public void tamanio() {
        IO.out.println(">>> " + pila.tamanio());
```

```
    }

    public static void main(String[] args) {
        new PilaTest();
    }
}
```

La clase IO, mediante el método *addController()*, crea un botón por cada método que aparezca en la prueba. Estos métodos devuelven *void* y no pueden tener parámetros. Así, podremos probar de una manera gráfica todos los métodos de nuestra clase.

Herencia y polimorfismo

Herencia

La herencia es uno de los mecanismos fundamentales de la programación orientada a objetos, por medio del cual una clase se construye a partir de otra. Una de sus funciones más importantes es proveer el polimorfismo.

La herencia relaciona las clases de manera jerárquica; una clase padre, superclase o clase base sobre otras clases hijas, subclases o clase derivada. Los descendientes de una clase heredan todos los atributos y métodos que sus ascendientes hayan especificado como heredables, además de crear los suyos propios.

En Java, sólo se permite la herencia simple, o dicho de otra manera, la jerarquía de clases tiene estructura de árbol. El punto más alto de la jerarquía es la clase *Object* de la cual derivan todas las demás clases. Para especificar la superclase se realiza con la palabra *extends*; si no se especifica se hereda de *Object*. Algunos ejemplos son:

```
public class Punto {...}                 //se herada de Object
public class Punto extends Object {...} //es lo mismo que la anterior
public class Punto3D extends Punto {...}
```

Cuando decimos que se heredan los miembros (atributos y métodos) de la superclase, quiere decir que se pueden referenciar con respecto a la clase hija. Por ejemplo, si la clase *SubClase* hereda de *SuperClase*, y esta tiene un método llamado *mtd1*, el siguiente mensaje es correcto:

```
      subClase.mtd1();
```

En este caso, la clase hija recibe el mensaje, y esta dispone de la funcionalidad por la herencia de la clase padre. Para que un atributo o método, sea directamente utilizable por la clase hija, debe ser *public*, *protected* o *friendly* (hasta ahora sólo hemos hablado de *public*, en el tema de paquetes aclararemos estos calificadores). Los miembros que sean *private*, no se podrán invocar directamente mediante *this*, pero su funcionalidad esta latente en la herencia.

Para referirnos a los miembros de la clase padre lo realizaremos mediante la palabra *super*. Con "*super.*" accedemos a los miembros y con "*super( )*" accedemos a los constructores.

Los constructores no se heredan, aunque existe llamadas implícitas a los constructores de las clases padre. Al crear una instancia de una clase hija, lo primero que se hace es llamar al constructor de la clase padre para que se inicialicen los atributos de la clase padre, sino se especifica, se llama al constructor sin parámetros. Por ello, si se quiere invocar otro constructor, la primera sentencia del constructor de la clase hija debe ser la llamada al constructor de la clase padre.

Veamos un ejemplo completo. Vamos a partir de la superclase Punto, que si no se especifica su superclase, hereda de *Object*.

```
public class Punto {
   private int x, y;

   public Punto(int x, int y) {
      this.setX(x);
      this.setY(y);
   }
```

```
   public Punto() {this(0, 0);}

   public final void setX(int x) {this.x = x;}
   public final void setY(int y) {this.y = y;}
   public int getX() {return this.x;}
   public int getY() {return this.y;}

   public double modulo() {
      return Math.sqrt(this.x * this.x + this.y * this.y);
   }
}
```

Ahora queremos crear una clase Punto pero que tenga en cuenta el tiempo, le llamaremos *PuntoTiempo*.

PuntoTiempo.java

```
public class PuntoTiempo extends Punto {
   private int t;

   public PuntoTiempo() {
      this(0,0,0) ;
   }
   public PuntoTiempo(int x, int y, int t) {
      super(x, y); // Si no se especifica, es: super();
      this.setT(t);
   }

   public int getT() {
      return this.t;
   }
   public final void setT(int t) {
      this.t = t;
   }
   public double velocidad() {
      return this.modulo() / this.getT();
   }
}
```

Cuando creamos una instancia de *PuntoTiempo*, se crea una instancia que lleva la funcionalidad de *Object*, *Punto* y lo añadido en *PuntoTiempo*. En la clase *PuntoTiempo*, en el constructor sin parámetros, como no hemos especificado el constructor de la superclase, se llama al *super()*.

COMPATIBILIDAD ENTRE VARIABLES DE CLASES

Una variable de tipo *MiClase*, puede contener instancias de *MiClase* o de cualquier subclase. En definitiva, una variable de tipo *Object*, puede contener cualquier instancia. En cambio, está prohibido que una variable tenga instancias de las superclases.

En el ejemplo anterior, una variable de tipo *Punto*, puede referencia instancias de tipo *PuntoTiempo*, pero una variable de tipo *PuntoTiempo* **NO** puede tener instancia de la clase *Punto*.

También se puede realizar casting entre clases. Mostremos un ejemplo con código Java:

```
      Punto p = new PuntoTiempo(1,2,3);
      //Error: PuntoTiempo pt = new Punto(3,2);
      p.setX(3);
      //Error con p.setT(3);
      ( (PuntoTiempo)p ).setT(3); // Error con (PuntoTitmpo)p.setT(3);
```

REDEFINICIÓN DE MÉTODOS

A veces, los métodos heredados de la superclase no son adecuados para la subclase. En este caso, debemos redefinir (sobrescribir) el método.

Supongamos que queramos realizar una clase *Punto3D* que hereda de *PuntoTiempo* para añadirle una tercera dimensión. En este caso, el método heredado *modulo* ya no es válido y lo debemos redefinir. Es recomendable añadir la directiva *@Override* para asegurarnos que no realizamos una sobrecarga por error. A continuación mostramos el código Java.

```
Punto3D.java
public class Punto3D extends PuntoTiempo {
   private int z;
   public Punto3D(int x, int y, int z, int t) {
      super(x, y, t);
      this.setZ(z);
   }

   public int getZ() {
      return this.z;
   }
   public final void setZ(int z) {
      this.z = z;
   }

   @Override
   public double modulo() {
      return Math.sqrt(this.getX() * this.getX() +
                   this.getY() * this.getY() + this.z * this.z);
   }
}
```

Fijarse que si enviamos un mensaje al método velocidad de una instancia de la clase Punto3D, esta se resuelve por la herencia de *PuntoTiempo*, cuando desde esa instancia se invoca al método *modulo*, se lanza la nueva versión redefinida en *Punto3D*.

CLASE OBJECT

Esta clase es la superclase de todas. Algunos de sus métodos son:

- *public String **toString**()*. Este método devuelve una cadena con la descripción de la clase. Es importante redefinir este método para adaptarlo a nuestra descripción, es muy útil en los procesos de depuración. Por ejemplo, en la clase *Punto* podría devolver los valores x e y entre paréntesis y separados por una coma: [3,5]. Se recomienda que siempre se redefina este método. Su código quedaría:

```
   @Override
   public String toString() {
      return "[" + this.getX() + "," + this.getY() + "]";
   }
```

Otra posible solución sería:

```
   @Override
   public String toString() {
      return "Punto[" + this.getX() + "," + this.getY() + "]";
   }
```

También, esta es una alternativa:

```
   @Override
   public String toString() {
      return "Punto[x:" + this.getX() + ",y:" + this.getY() + "]";
   }
```

- *public boolean **equals**(Object obj)*. Este método devuelve un boolean indicando si la instancia actual es igual a la que se pasa por parámetro. En *Object* compara direcciones, devolviendo true si son la misma dirección. Normalmente, esta no es la implementación necesitada. Por ejemplo, en la clase Punto debemos considerar que dos instancias de punto son iguales si la coordenada x e y coinciden. Su código quedaría:

```
   @Override
   public boolean equals(Object obj) {
       boolean result;
       if (obj == null || this.getClass() != obj.getClass()) {
           result = false;
       } else {
          final Punto otro = (Punto) obj;
          result = otro.getPtX() == this.ptX;
       }
       return result;
```

```
    }
```

- *public int **hashCode**()*. Devuelve un código hash de este objeto. Normalmente, si se sobrescribe ***equals*** se debe sobre escribir ***hashcode***. Se debe buscar que el código hash sea diferente entre dos instancias que no se consideren iguales.
- *protected void **finalize**() throws Throwable*. Este es el método destructor, se debe redefinir si queremos realizar alguna acción para liberar recursos. Cuando un objeto queda huérfano, el recolector de basura lo destruye; pero esta acción no es inmediata. Si queremos que actúe el recolector, lo podemos lanzar mediante la clase System*:* `System.gc()`. Un ejemplo sería:

```
    @Override
    protected void finalize() throws Throwable {
        System.out.println("Se liberan recursos");
        super.finalize();
    }
```

- *protected Object **clone**() throws CloneNotSupportedException*. Este método nos proporciona una copia de la instancia, pero para ello, la clase a duplicar debe implementar el interface *Cloneable* (no tiene ningún método). Nuestra clase clone, debe encargarse de duplicar todas las instancias que lo formen. En un apartado posterior, veremos un ejemplo completo.

FINAL

Si lo aplicamos a una clase, no se permite que existan subclases.

Si lo aplicamos a un método, no se permite que subclases redefinan el método.

Clases abstractas

Una clase abstracta es una clase que no se puede instanciar, pero si heredar. También, se pueden definir constructores. Se utilizan para englobar clases de diferentes tipos, pero con aspectos comunes. Se pueden definir métodos sin implementar y obligar a las subclases a implementarlos.

Por ejemplo, podemos tener una clase *Figura*, que representa una figura general, y subclases con figuras concretas (*Triangulo*, *Circulo*...). Podemos establecer métodos comunes como *dibujar*, que sin conocer el tipo de figura, sea capaz de dibujarla. Pero para que esto funcione correctamente, cada figura concreta debe implementar su versión particular de *dibujar*.

Si añadimos el calificador *abstract* después del calificativo *public* del método, le convertimos en abstracto y no hace falta que lo implementemos, basta con definirlo. Si añadimos *abstract* después del calificativo *public* de la clase, la convertimos en clase abstracta; esta acción es obligatoria si algún método es abstracto.

La ventaja en utilizar clases abstractas, es que podemos trabajar con variables o parámetros de tipo *Figura* y llamar a los métodos comunes sin saber a priori el tipo concreto de *Figura*. Esto permite en el futuro, añadir nuevas *Figuras*, sin cambiar las clases ya desarrolladas. A este concepto se le llama polimorfismo.

El polimorfismo es el modo en que la POO implementa la polisemia, es decir, un solo nombre para muchos conceptos.

Este tipo de polimorfismo se debe resolver en tiempo de ejecución y por ello hablamos de polimorfismo dinámico, a diferencia del polimorfismo estático que se puede resolver en tiempo de compilación. Recordemos que éste último se implementaba mediante la sobrecarga de los métodos.

A continuación presentamos un ejemplo mediante notación UML (Lenguaje Unificado de Modelado) y en Java

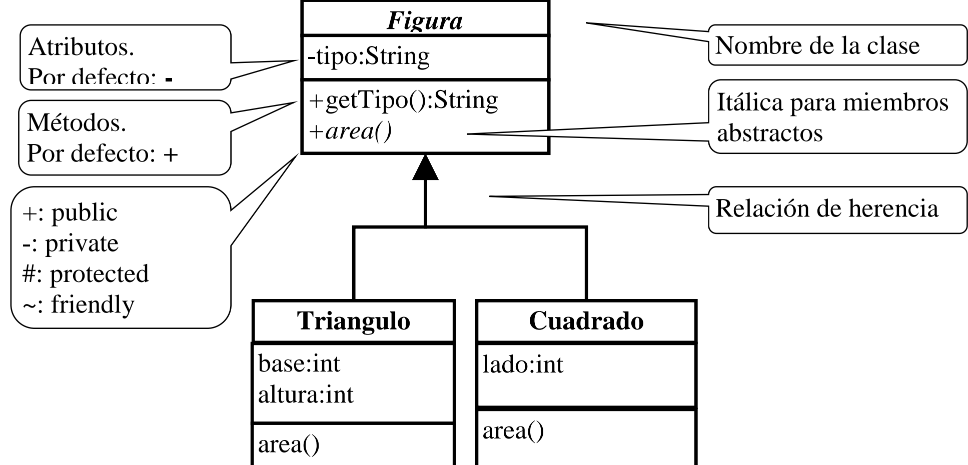


Figura.java

```
public abstract class Figura {
   private String tipo;
   public Figura(String tipo) {
      this.tipo = tipo;
   }
   // getters & setters
   public abstract double area();
}
```

Triangulo.java

```
public class Triangulo extends Figura {
   private int base, altura;
   public Triangulo(String tipo, int base, int altura) {
      super(tipo);
      this.setBase(base);
      this.setAltura(altura);
   }
   // getters & setters
   @Override
   public double area() {
      return (double) this.base * this.altura / 2;
   }
}
```

Cuadrado.java

```
public class Cuadrado extends Figura {
   private int lado;
   public Cuadrado(String tipo, int lado) {
      super(tipo);
      this.setLado(lado);
   }
   // getters & setters
   @Override
   public double area() {
      return (double) this.lado * this.lado;
   }
}
```

Polimorfismo.java

```
...
   public void polimorfismo(Figura una){
      System.out.println("Tipo: " + una.getTipo());
      System.out.println("Area: " + una.area());
   }
...
```

✍ *Crear las clases Figura, Triangulo y Cuadrado*

Interfaces

Un interface en Java es un tipo abstracto que representa un conjunto de métodos que las clases pueden implementar y un conjunto de constantes estáticas, es similar a una clase abstracta pura. Igual que las clases abstractas, los interfaces no se puede instanciar. Su utilidad es crear variables de tipo interface e independizarse de las clases que contiene.

Para definir un interface se realiza de forma similar a una clase abstracta pero con la palabra *interface* en lugar de *class*. Pueden existir variables o parámetros de tipo interface, pero sólo pueden contener instancias de clases que hayan implementado el interface.

Una clase puede implementar uno o varios interfaces, para ello se utiliza la palabra *implements*.

Veamos un ejemplo.

IBox.java

```
public interface IBox {
   public void put(Object objeto);
   public Object get();
}
```

CBox.java

```
public class CBox implements IBox {
   private Object objeto;

   public CBox(){
      this.objeto=null;
   }

   @Override
   public Object get() {
      return this.objeto;
   }

   @Override
   public void put(Object objeto) {
      this.objeto = objeto;
   }
}
```

Los interfaces pueden heredar de otros mediante la palabra *extends*, además se permite herencia múltiple.

Mediante interfaces también se aporta el polimorfismo, de hecho, una clase abstracta con todos sus métodos abstractos es similar a un interface.

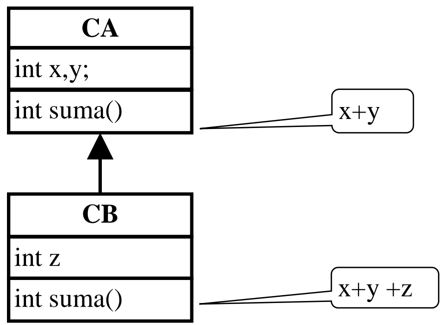
En general, es mejor utilizar interface; sólo cuando se necesite añadir código, se debe utilizar las clases abstractas.

Compatibilidad de variables

Una variable de tipo interface puede contener instancias de cualquier clase que implemente el interface.

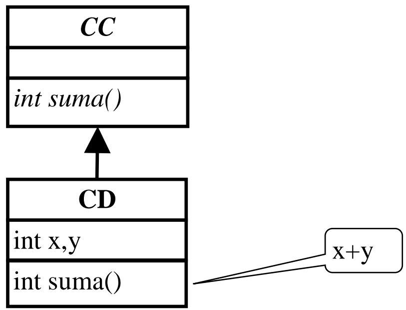
Ejercicios

✍ ***Implementar las clases CA y CB, donde CA hereda de CB***



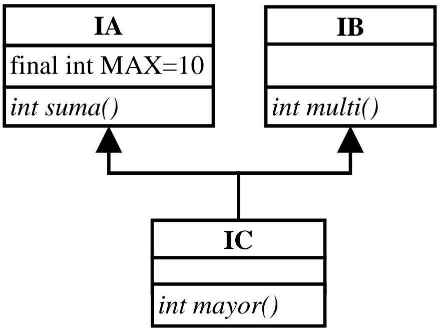
✍ **Dado el código:** ***CA a = new CB(1,2,3);*** **¿qué valor devuelve "*a.suma();*"?**

✍ ***Implementar las clases CC y CD***



✍ ***Añadir e implementar el método int dobleSuma() en CC***

✍ ***Definir el interface IA, IB e IC***



✍ ***Ampliar CA para que implemente IC***

✍ ***interface Comparable, método compareTo(Object uno, Object dos) (si uno es <, == o > que dos, devuelve -1, 0, +1 respectivamente***

✍ ***Ampliar CA para que implemente Comparable***

Colaboración entre clases

Llegados a este punto, se recomienda pasar a un entorno de desarrollo avanzado, como por ejemplo Eclipse. En el área de documentación del MIW se encuentra un tutorial rápido sobre Eclipse.

Relación de clases

Una aplicación orientada a objetos es una colección de objetos, algunos jerárquicos, que colaboran entre sí para dar una funcionalidad global. Cada objeto debe asumir una parte de las responsabilidades y delegar otras a los objetos colaboradores.

Existirá un objeto que asume las funciones de la aplicación principal, será el encargado de crear las instancias oportunas y tener un único punto de entrada (*main*).

Cabe hacernos una pregunta fundamental, ¿en cuántos objetos debemos descomponer la aplicación? Si realizamos objetos muy pequeños, cada uno de ellos será fácil de desarrollar y mantener, pero resultará difícil la integración. Si realizamos objetos grandes, la integración será fácil, pero el desarrollo de los objetos complicado. Se debe buscar un equilibrio.

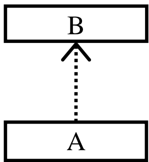
En definitiva, y sin adentrar más en la estructuración de las clases de una aplicación, nuestra aplicación tendrá un conjunto de clases que están relacionadas entre ellas.

Se considera que existe una **relación entre clases** si sus objetos colaboran, es decir, se mandan mensajes. Tenemos, principalmente, tres tipos de relaciones entre clases: uso, agregación y composición, aparte de la herencia que ya vimos.

Relación de Uso

Se dice que existe una relación de uso entre la clase A y la clase B, si un objeto de la clase A lanza mensajes al objeto de la clase B, para utilizar sus servicios.

Es una relación que se establece entre un cliente y un servidor, en un instante dado, a través de algún método. No es una relación duradera, sino esporádica.

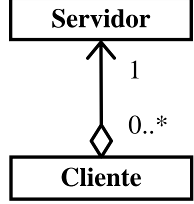



```
public class A {
   ...
   public void metodo(B b){
       b.servicio();
   }
   ...
}
```

Relación de agregación

Se dice que la clase A tiene una relación de agregación con la clase B, si un objeto de la clase A delega funcionalidad en los objetos de la clase B.

El ciclo de vida de ambos no coincide. La clase A delega parte de su funcionalidad, pero no crea el objeto B. Varios objetos pueden estar asociados con el objeto B. Al objeto A, le deben pasar la referencia del objeto B. Es una relación duradera entre cliente (A) y servidor (B), en la vida del cliente.




```
public class Cliente {
   private Servidor servidor;
   public Cliente(Servidor servidor) {
      this.servidor = servidor;
```

```
    }
    ...
}
```

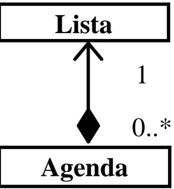
```
public class Servidor {
   ...
   public String servicio() {
      return "Servido";
   }
   ...
}
```

RELACIÓN DE COMPOSICIÓN

Se dice que existe una relación de composición entre la clase A y la clase B, si la clase A contiene un objeto de la clase B, y delega parte de sus funciones en la clase B.

Los ciclos de vida de los objetos de A y B coinciden. El objeto A es el encargado de crear la instancia de B y de almacenarla, siendo su visibilidad privada. Cuando se destruye A, también se destruye B.

Por ejemplo, se pretende realizar una clase llamada Agenda y para ello se apoya en la clase Lista, es decir, dentro de la clase Agenda se crea una instancia de la clase Lista, y algunas de las funcionalidades que aporta la clase Agenda se realizan con los métodos de la clase Lista.



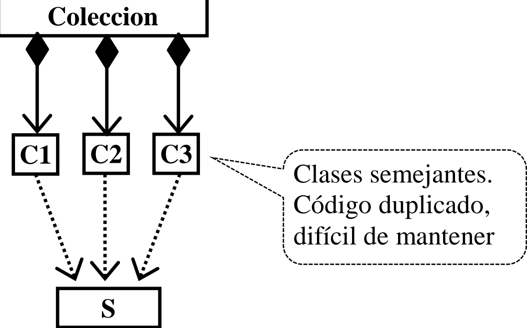

```
public class Agenda {
   private Lista lista;

   public Agenda() {
      this.setLista(new Lista());
   }
   ...
}
```

Beneficio de la herencia

Es importante realizar un uso de la herencia para optimizar el código y evitar las redundancias.

Ejemplo incorrecto. Veamos un mal uso de las clases, donde aparece código redundante y un mal uso de la herencia. Resulta incorrecto que el mensaje lo interprete el emisor




```
      //MAL DISEÑO
      public class Coleccion{
      ...
         for(C item: miColeccion){
```

```
                if (item.getName.equals("C1"){...procesar C1}
                else if (item.getName.equals("C2"){...procesar C2}
                ...
        }
```

Ejemplo correcto. Buen uso de la herencia. El mensaje lo interprete el receptor

Coleccion
C
S
C1
C2
C3
Código común factorizado

```
    //BUEN DISEÑO
    public class Coleccion{
    ...
       for(C item: miColeccion){
            item.procesar();
       }
```

Beneficio de los interfaces

En una relación de agregación entre la clase A y la clase B, la clase A debe lanzar mensajes a la clase B, y para ello es necesario que a la clase A le pasen la referencia de la clase B.

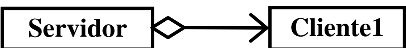
Una vez implementada la clase A, cualquier cambio en la clase B (por ejemplo, una mejora o cambio en la implementación de B) provoca que debiéramos cambiar todas las referencias de la clase A sobre la B. Incluso puede que cuando se desarrolla la clase A, no se dispone de la implementación de la clase B.

Para ello disponemos de dos mecanismos:

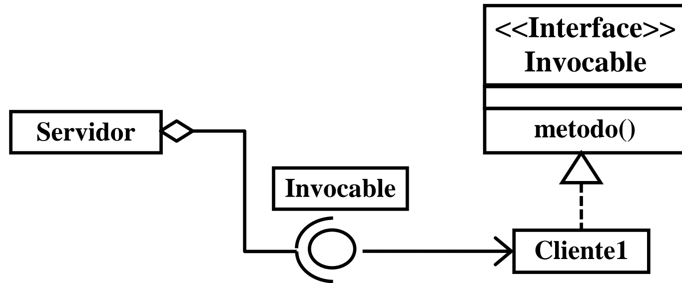
- Mediante herencia de clases. En este caso, la clase A referencia los métodos de una superclase B, pudiendo existir varias subclases de B que sirvan a tal fin.
- Mediante interfaces. Aquí, la clase A referencia los métodos de un interface, pudiendo existir un conjunto de clases que implementan ese interface.

Fijarse, que en ambos casos, cualquier cambio en las implementaciones de B no afectan al código de la clase A.

Veamos un ejemplo concreto. En esta asociación existe una dependencia clara entre la clase *Servidor* y la clase *Cliente*.



Pero en este ejemplo tenemos un problema, ya que se pretende construir un servidor para que de servicio a muchos clientes y se debe romper esa dependencia. Obligar a los clientes a heredar de una clase no es una buena solución, ya que le limitamos en su diseño. La mejor solución es utilizar los interfaces. A continuación presentamos el nuevo diseño:



En este nuevo diseño, hemos roto la dependencia del servidor respecto a las diferentes implementaciones de los clientes.

Veamos el código en Java:

Invocable.java

```
public interface Invocable {
   public void newMsg(String msg);
}
```

Servidor.java

```
public class Servidor {
   private Invocable cliente;

   public void add(Invocable cliente) {
      this.cliente = cliente;
   }

   public void remove() {
      this.cliente = null;
   }

   public void browcast() {
      if (this.cliente != null) cliente.newMsg("Nuevo mensaje");
   }
}
```

Cliente1.java

```
public class Cliente1 implements Invocable {
   public Cliente1(Servidor servidor) {
      servidor.add(this);
   }

   @Override
   public void newMsg(String msg) {
      System.out.println("Llega un nuevo mensaje..." + msg);
   }
}
```

Paquetes

Los paquetes agrupan un conjunto de clases que trabajan conjuntamente sobre el mismo ámbito. Es una facilidad ofrecida por Java para agrupar sintácticamente clases que van juntas conceptualmente y definir un alto nivel de protección para los atributos y los métodos.

La ventaja del uso de paquetes es que las clases quedan ordenadas y no hay colisión de nombres. Si dos programadores llaman igual a sus clases y luego hay que juntar el código de ambos, basta explicitar a qué paquete nos referimos en cada caso.

La forma de nombrar los paquetes es con palabras (normalmente en minúsculas) separadas por puntos, estableciendo una jerarquía de paquetes; la jerarquía de paquetes es independiente de la jerarquía de clases. Las clases deben almacenarse en una estructura de carpetas que coincidan con la estructura de paquetes.

Para que una clase se asocie a un paquete se realiza con la sentencia *package*, se debe situar antes de la clase. Algunos ejemplos de uso son:

```
public class Prueba ...

package p1.p2;
public class MiClase ...

package p1;
public class Clase2 ...
```

Tanto los ficheros **.java* como los ficheros **.class*, se deben almacenar en una jerarquía de carpetas que coincidan con los nombres de los paquetes. En la Figura 4. Paquetes en JavaFigura 4 se muestra cómo quedaría la jerarquía de carpetas.

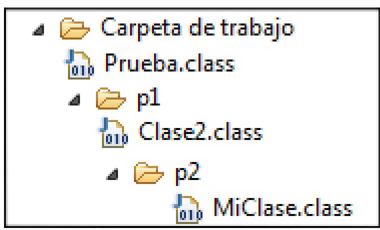


Figura 4. Paquetes en Java

Para su distribución, se genera un fichero *jar*, con la estructura de carpetas y clases. Recordar que este fichero no hace falta descomprimirlo. Para su utilización, se debe referenciar mediante la variable de entorno *CLASSPATH*.

Para referenciar cualquier clase en Java, se debe poner el paquete al que pertenece. Por ejemplo: *java.lang.String*, *java.lang.Integer*... Hasta ahora no ha sido necesario, porque el paquete *java.lang* se importa de forma automática.

Para evitar poner siempre la jerarquía de paquetes, se puede utilizar la sentencia *import*, con ello se puede importar un paquete entero (pero no los paquetes inferiores) o una clase.

Mostramos un ejemplo de uso:

```
public class Una{
   private java.util.Map mapa;

   public Una() {
      mapa = new java.util.HashMap();
      mapa.put("clave", "valor");
   }
   // ...
```

```
import java.util.Map;
import java.util.*;  //importa todas las clases del paquete, NO los subpaquetes.
public class Una{
   private Map mapa;
   // ...
```

Los paquetes permiten utilizar los modos de protección *protected* y *friendly*, que veremos en el siguiente apartado.

Nota. Se recomienda utilizar un IDE, como por ejemplo *Eclipse*. Para ejecutar una clase de un paquete desde la consola que no está referenciado mediante la variable de entorno CLASSPATH, nos debemos situar en la carpeta raíz de los paquetes y referenciar al fichero *class* mediante la ruta completa.

Un ejemplo es: `java paquete/MiClase`

Visibilidad

Hasta ahora hemos trabajado con dos tipos de visibilidad: *public* y *private*. Recordemos que un elemento público es visible para todos y privado sólo para uso interno.

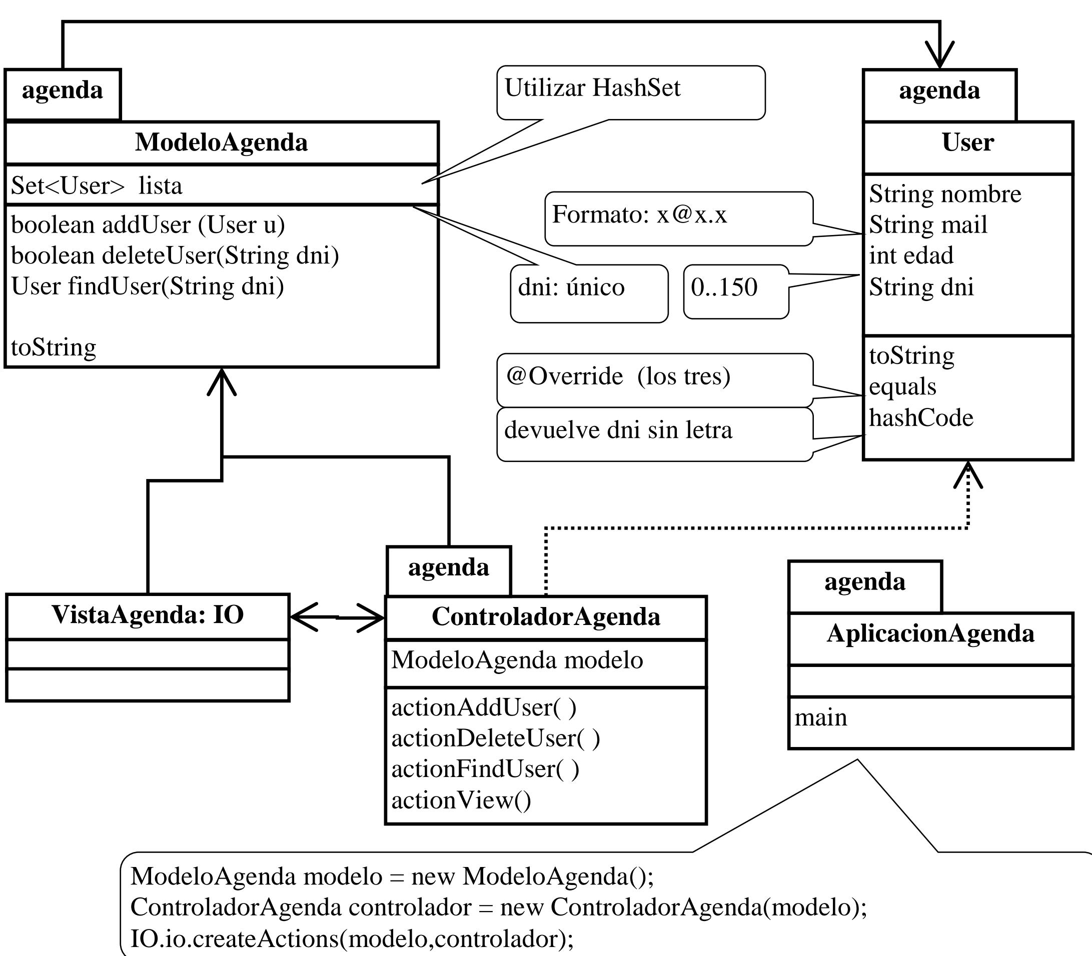
En Java se distinguen dos tipos más de visibilidad: *protected* y sin calificador (*friendly*).

Cuando calificamos a un elemento como *protected*, significa que es visible dentro del mismo paquete o cualquier subclase de forma directa (mediante *super.elemento*). Si pretendemos que una subclase acceda de forma directa a los atributos de la superclase, el calificador apropiado es *protected*.

Cuando dejamos a un elemento sin calificar (*friendly*), significa que sólo es visible dentro del mismo paquete.

✍ ***Probar las clases anteriores***

✍ ***AVANZADO. Implementar la aplicación AGENDA. Lleva una agenda de usuarios. Permite las acciones básicas: añadir, borrar, buscar y ver toda la agenda.***



Método de Desarrollo de Programas

Un programa orientado a objetos en ejecución es una colección jerárquica de objetos contenidos en otros objetos sucesivas veces que colaboran todos entre sí. Paralelamente, un programa en edición/compilación es una colección de las clases de estos objetos relacionadas entre sí.

Las funciones del ***Programa Principal*** las asumirá un único objeto que englobe todo el sistema de objetos (mediante un desencadenamiento de instanciaciones) al que lanzándole un único mensaje realice todas las funciones deseadas (mediante un desencadenamiento de mensajes). Fuera de ese objeto no puede existir nada, no tiene sentido que reciba parámetros, no tiene sentido devolver un valor a nadie.

La clase de este objeto ***Principal*** suele responder a nombres como Aplicación, Sistema, Gestor, Simulador, Juego, etc. El mensaje a ***Principal*** suele responder a nombres como ejecutar, gestionar, simular, jugar, etc. Por tanto, de forma genérica para cualquier programa orientado a objetos:

Cliente1.java
```
class Aplicacion {
    // ...
    public void ejecutar() {
        // ...
    }
    public static void main(String[] arg) {
        Aplicacion aplicacion = new Aplicacion();
        aplicacion.ejecutar();
    }
```

```
}
```

Una vez establecido el objeto ***Principal*** e inicial queda por determinar cuántos otros objetos existen en el programa: ¿hasta qué punto se realizan descomposiciones sucesiva de objetos en objetos que forman la jerarquía de composición, fundamental en un programa orientado a objetos? Dado el mismo conjunto de requisitos, más módulos significan menor tamaño individual del módulo. Sin embargo, a medida que crece el número de módulos, el esfuerzo asociado con la integración de módulos también crece. Estas características llevan a una curva de coste total. Hay un número, M, de módulos que resultaría en un coste de desarrollo mínimo, pero no tenemos la sofisticación necesaria para predecir M con seguridad.

Entonces, dado que no se puede establecer el número de distintas clases que tiene un programa, se recurre a la siguiente regla de desarrollo: aceptar una clase más si existen fundamentos para pensar que se reducen los costes de desarrollo totales que favorezca mayores cotas de abstracción, encapsulación, modularización y jerarquización que faciliten la legibilidad, mantenimiento, corrección, etc.

Una vez que se conoce de dónde partir (la clase del objeto ***Principal***) y hasta dónde llegar (hasta que nuevas clases no reduzcan los costes), resta saber qué camino seguir para el desarrollo del programa.

El método propuesto es una simplificación extrema y variación de HOOD (diseño orientado a objetos jerárquico). El resultado es un método muy sencillo y utilizable en pequeñas aplicaciones. Este método contiene 5 fases que se aplican cíclicamente para cada clase del programa:

1. Tomar una clase
2. Definir el comportamiento
3. Definir los atributos
4. Codificar los métodos
5. Recapitular nuevas clases y métodos.

A continuación se aplicará el método fase a fase bajo la siguiente nomenclatura: "F.C" siendo 'F' la fase actual (con valores de 1 a 5) y 'C' el ciclo de aplicación (con valores de 1 a N clases del programa). Posteriormente, se sintetizarán esquemáticamente todas las fases del método.

El programa a resolver debe permitir a dos usuarios jugar a las "Tres En Raya" (TER):

- Inicialmente el tablero estará vacío
- Los jugadores pondrán las fichas alternativamente, siendo posible alcanzar las "Tres en Raya" con la quinta puesta
- En caso contrario, una vez puestas las 6 fichas, se moverán las fichas alternativamente
- Tanto en las puestas como en los movimientos, se deben requerir las coordenadas a usuario validando la corrección de la puesta/ movimiento
- Se debe detectar automáticamente la finalización del juego al lograrse "Tres en Raya": tras la quinta y sexta puesta y tras cada movimiento

(1.1) Tomar una clase: al principio de todo el desarrollo del programa será la clase del objeto que engloba todo el sistema: *TER*.

```
class TER {
    public static void main(String[] arg) {
        TER partida = new TER();
        partida.jugar();
    }
}
```

(2.1) Determinar el comportamiento: al principio de todo el desarrollo del programa será el método de la clase que engloba todo el sistema que dispare toda la funcionalidad deseada -jugar-: no tendrá parámetros y no devolverá nada; el juego es "todo" el universo de objetos.

```
class TER {
    public void jugar() {
        // ..
    }
...
```

(3.1) Determinar los atributos con sus respectivas clases; no es estrictamente necesario determinar todos y cada uno de los atributos, sino que se deben establecer aquellos que sean "obvios" en el momento; si se omitiese alguno surgirá su necesidad en el siguiente punto (4.1).

Los atributos soportarán los datos que posibiliten el proceso del juego: memorizar dónde están las fichas puestas/movidas, saber a qué jugador le corresponde poner/mover y su color de fichas, etc.

En programación orientada a objetos, se recurre a atributos que son objetos de otras clases responsables de los detalles del soporte y manipulación de esos datos;

Para el programa de las "Tres en Raya":

- Se crea un objeto de la clase Tablero; responsable de aparecer vacío, guardar las fichas que se pongan, detectar las "Tres en Raya", mostrar la situación de las fichas, etc.
- Se crean dos objetos de la clase Jugador; responsables de memorizar el color de cada uno, comunicarse con cada usuario y controlar automáticamente sus puestas y movimientos, etc.
- Se crea un objeto de la clase Turno; responsable de memorizar a qué jugador le corresponde poner/mover, alternar el jugador, etc.

```
public class TER {
    private final Tablero tablero = new Tablero();
    private final Jugador[] jugadores = new Jugador[2];
    private final Turno turno = new Turno();

    public TER() {
        jugadores[0] = new Jugador('o');
        jugadores[1] = new Jugador('x');
    }

    public void jugar() {
        // ...
    }

    public static void main(String[] arg) {
        TER partida = new TER();
        partida.jugar();
    }
}
```

(4.1) Codificar métodos: no debe ser un impedimento incluir en la codificación de los métodos el paso de mensajes correspondientes a métodos que no se encuentren definidos previamente en sus correspondientes clases. Hay que codificar los métodos de la clase actual, TER, delegando tareas a los objetos de las clases de los atributos, Tablero, Jugador y Turno.

```
    public void jugar() {
        tablero.mostrar();
        for (int i = 0; i < 5; i++) {
            jugadores[turno.toca()].poner(tablero);
            turno.cambiar();
            tablero.mostrar();
        }
        if (tablero.hayTER()) {
            jugadores[turno.noToca()].victoria();
        } else {
            jugadores[turno.toca()].poner(tablero);
            tablero.mostrar();
            while (!tablero.hayTER()) {
                turno.cambiar();
                jugadores[turno.toca()].mover(tablero);
                tablero.mostrar();
            }
            jugadores[turno.toca()].victoria();
        }
    }
```

(5.1) Recapitular: nuevas clases aparecidas en los puntos (2.1), (3.1) y (4.1) y métodos correspondientes a los mensajes enviados a objetos de estas clases en el punto (4.1). El valor devuelto y parámetros de cada método se extraen del contexto del envío del mensaje.

La recapitulación abre de nuevo la aplicación de las 5 fases del método para cada una de las clases, pero con los siguientes matices:

- La primera fase, tomar una clase, que "anulada" al limitarse a escoger una de las clases recapituladas anteriormente. Se recomienda que la elección sea la que inspire más riesgos en su implantación
- La segunda fase, determinar el comportamiento público de clases anteriores queda "anulada" por la quinta fase de recapitulación anterior

(1.2) Tomar una clase: ***Jugador*** de la fase (5.1).

(2.2) Determinar el comportamiento: de la fase (5.1).

(3.2) Determinar los atributos con sus respectivas clases; para poner, mover y cantar victoria un Jugador solo precisa saber el color de sus fichas.

```
public class Jugador {
    private char color;
    public Jugador(char c) {
        // TODO Auto-generated constructor stub
    }
    public void poner(Tablero tablero) {
        // TODO Auto-generated method stub
    }

    public void victoria() {
        // TODO Auto-generated method stub
    }

    public void mover(Tablero tablero) {
        // TODO Auto-generated method stub
    }
}
```

(4.2) Codificar métodos: no debe ser un impedimento incluir en la codificación de los métodos nuevas clases "auxiliares" no previstas hasta el momento.

```
public class Jugador {
    private char color;
    public Jugador(char color) {
        this.color = color;
    }
    public void poner(Tablero tablero) {
        IO.out.println("juega: " + color);
        Coordenada destino = new Coordenada();
        do {
            destino.recoger("Coordenada destino de puesta");
        } while (!destino.valida() || tablero.ocupado(destino));
        tablero.poner(destino, color);
    }
    public void mover(Tablero tablero) {
        IO.out.println("juega: " + color);
        Coordenada origen = new Coordenada();
        do {
            origen.recoger("Coordenada origen de movimiento");
        } while (!origen.valida() || tablero.vacio(origen));
        tablero.sacar(origen);
        tablero.mostrar();
        this.poner(tablero);
    }
    public void victoria() {
        IO.out.println("las " + color + " han ganado y ...");
    }
}
```

Esta solución opta por la inclusión de una nueva clase Coordenada que aglutine el tratamiento de la fila y la columna de una casilla del tablero donde poner/mover fichas.

(1.3) Tomar una clase: Tablero de la fase (5.1)

(2.3) Determinar el comportamiento: de las fases (5.1) y (5.2)

(3.3) Determinar los atributos con sus respectivas clases; para poner, sacar y detectar si *hayTER* o si una casilla está ocupado o vacía un Tablero solo precisa soportar 3x3 caracteres que memoricen las posiciones de las fichas (el carácter '-' representa la ausencia de fichas).

(4.3) Codificar métodos.

```java
public class Tablero {
     private final char[][] fichas = new char[3][3];
     private static final char VACIO = '-';
     public Tablero() {
          for (int i = 0; i < 3; i++) {
               for (int j = 0; j < 3; j++) {
                    fichas[i][j] = VACIO;
               }
          }
     }
     public void mostrar() {
          IO.out.clear();
          for (int i = 0; i < 3; i++) {
               for (int j = 0; j < 3; j++) {
                    IO.out.print(fichas[i][j]);
                    IO.out.print(' ');
               }
               IO.out.println();
          }
          IO.out.println();
     }

     public boolean hayTER() {
          return this.hayTER('o') || this.hayTER('x');
     }

     private boolean hayTER(char color) {
          boolean victoria = false;
          int diagonal = 0;
          int inversa = 0;
          int[] filas = new int[3];
          int[] columnas = new int[3];

          for (int i = 0; i < 3; i++) {
               filas[i] = 0;
               columnas[i] = 0;
               for (int j = 0; j < 3; j++) {
                    if (fichas[i][j] == color) {
                         if (i == j) {
                              diagonal++;
                         }
                         if (i + j == 2) {
                              inversa++;
                         }
                         filas[i]++;
                         columnas[j]++;
                    }
               }
          }
          if ((diagonal == 3) || (inversa == 3)) {
               victoria = true;
          } else {
               for (int i = 0; i < 3; i++) {
                    if ((columnas[i] == 3) || (filas[i] == 3)) {
                         victoria = true;
                    }
               }
          }
          return victoria;
     }
     public boolean ocupado(Coordenada coordenada) {
          return fichas[coordenada.getFila()][coordenada.getColumna()] != VACIO;
     }
     public boolean vacio(Coordenada coordenada) {
          return (!this.ocupado(coordenada));
     }
     public void poner(Coordenada coordenada, char color) {
          fichas[coordenada.getFila()][coordenada.getColumna()] = color;
     }
     public void sacar(Coordenada coordenada) {
          this.poner(coordenada, VACIO);
     }
}
```

(5.3) Recapitular

(1.4) Tomar una clase: Coordenada de la fase (5.2).

(2.4) Determinar el comportamiento: de las fases (5.2) y (5.3).

(3.4) Determinar los atributos con sus respectivas clases; para recoger, validar y obtener fila y columna de una coordenada, solo se precisa memorizar la fila y la columna.

(4.3) Codificar métodos:

```
public class Coordenada {
    private int fila;
    private int columna;

    public void recoger(String titulo) {
        fila = IO.in.readInt("Dame una fila");
        columna = IO.in.readInt("Dame una columna");
    }

    public boolean valida() {
        return fila <= 2 && fila >= 0 && columna <= 2 && columna >= 0;
    }

    public int getFila() {
        return fila;
    }

    public int getColumna() {
        return columna;
    }
}
```

(5.4) Recapitular: no surgen ni nuevas clases ni nuevos métodos de clases ya recapituladas.

(1.5) Tomar una clase: Turno de la fase (5.1)

(2.5) Determinar el comportamiento: de las fases (5.1)

(3.5) Determinar los atributos con sus respectivas clases; para cambiar y obtener a quien toca y el contrario solo se precisa memorizar un valor alternativo entre 1 y 2.

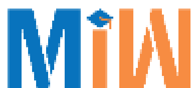
(4.5) Codificar métodos:

```
public class Turno {
    private int valor = 0;

    public void cambiar() {
        this.valor = (this.valor + 1) % 2;
    }

    public int toca() {
        return valor;
    }

    public int noToca() {
        return (valor + 1) % 2;
    }
}
```



2.

Programación avanzada

Excepciones

Una excepción es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias. Hasta ahora, hemos visto que controlamos el flujo de ejecución mediante las sentencias de control (*if*, *while*...). Con la gestión de excepciones, Java nos ofrece una técnica nueva que nos facilita la programación.

Ya no hará falta preguntar a lo largo de nuestro programa si ha existido algún error; cuando se produce una excepción, se lanza automáticamente el código que la procesa. Pudiendo separar la programación normal, con la programación de tratamiento de errores.

Cuando un método lanza una excepción, se crea un objeto de la clase o subclases de *Exception*, que contiene los detalles de la excepción. Este objeto contiene información de las circunstancias en que se provocó y la pila de llamadas entre los métodos. El sistema busca hacia atrás en la pila de llamadas, empezando por el método en el que ocurrió el error, hasta que encuentra un método que contiene el *manejador de excepción* adecuado.

Muchas clases de errores pueden utilizar excepciones, desde problemas de hardware, como la avería de un disco duro, a los simples errores de programación, como tratar de acceder a un elemento de un array fuera de sus límites.

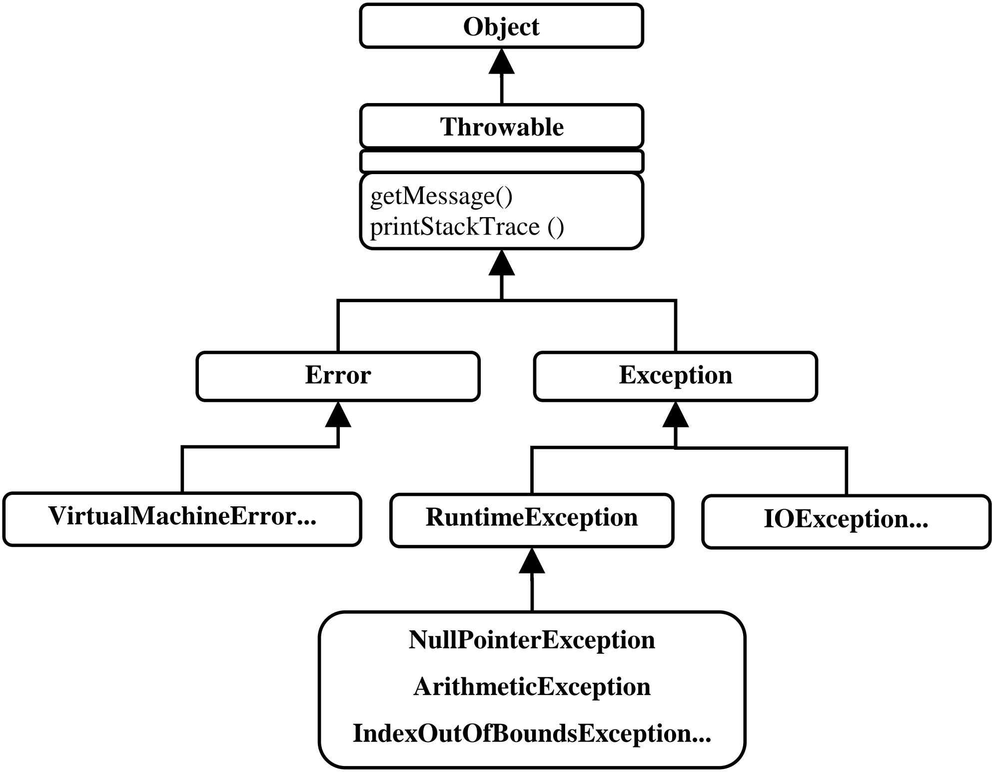
Podemos agrupar las excepciones en tres tipos fundamentales:

- **Error**. Excepciones que indican problemas muy graves asociados al funcionamiento de la máquina virtual y suelen ser no recuperables. Normalmente, casi nunca deben ser capturadas. Un ejemplo sería que la máquina virtual de Java se ha quedado sin memoria del sistema, o algún problema con el hardware.
- **Exception**. Excepciones que se utilizan para indicar que un mensaje lanzado a un objeto no ha podido realizar su acción. Es una filosofía de programación, se pretende asegurar que el origen del mensaje es consciente de que puede haber problemas. Un ejemplo sería que se intenta acceder a un fichero que no existe.
- **RuntimeException**. Aunque esta clase es una subclase de *Exception*, tiene un tratamiento especial. Es una Excepción asociada a una mala programación, por ejemplo que un índice está fuera del rango del array. Estas excepciones no deben ser capturadas, sino que se debe corregir el problema de programación.

Cuando en la ejecución de un método se produce una excepción, se genera una instancia de una subclase de *Throwable* que describe el tipo de excepción. De forma inmediata se aborta la ejecución del método y se mira si el método tiene definido un manejador de excepciones. Si lo tiene definido, se trata la excepción y se continua normalmente, sino, se lanza la excepción al método invocador del actual.

Repitiendo este proceso hasta llegar al método *main* inicial. Finalmente, si el método *main* no lo trata, se aborta la ejecución de la aplicación Java.

A continuación, presentamos parte de la jerarquía de clases que heredan de *Throwable*.



La gestión de excepciones tiene dos partes bien diferenciadas:

- Manejo de excepciones
- Lanzamiento de excepciones, pudiéndola realizar el programador mediante sentencias especiales o por el sistema

Manejo de excepciones

Cuando el programador va a ejecutar una porción de código que pueda provocar una excepción, puede actuar de dos maneras: tratando la excepción o pasársela al método invocador. Para tratar la excepción, se debe incluir el código dentro de un bloque *try-catch*, para pasársela al método invocador se añade la palabra reservada *throws* a la definición del método.

Hay que dejar claro que las excepciones causadas por una mala programación (por ejemplo, un índice fuera de límites) se deben corregir y no se deben tratar. Las excepciones que se tratan son aquellas que se lanzan por ser parte del funcionamiento previsto y controlado de un objeto, por ejemplo, error en la lectura de un fichero.

La gramática de un *try-catch* es la siguiente:

```
int[] array = new int[10];
try {
   array[11] = Integer.parseInt("12a");
   new FileReader("fichero");
   // ...
} catch (NumberFormatException nfe) {
   // Tratamiento de excepción
   nfe.printStackTrace(); // se imprime el traceado de pila
} catch (RuntimeException re) {// igual o más general que la anterior
   System.out.println(re.getMessage());
} catch (Exception e) { // igual o más general que la anterior
   System.out.println(e.getMessage());
} finally {
```

```
        System.out.println("Siempre se ejecuta");
    }
```

En el ejemplo anterior, realmente no hemos incorporado ningún tratamiento especial, simplemente se imprime un mensaje para comprobar el funcionamiento de la gramática.

En el ejemplo siguiente, se le pide al cliente un valor de tipo numérico, si no se puede convertir a *int*, se le vuelve a pedir hasta que se pueda convertir.

```
    int dato = 0;
    boolean isInt = false;
    while (!isInt) {
       try {
          dato = Integer.parseInt(IO.in.read("Dame un dato entero"));
          isInt = true;
       } catch (NumberFormatException nfe) {
          IO.out.println("Error en la lectura del dato");
       }
    }
    IO.out.println("Correcto, dato leido: " + dato);
```

El funcionamiento es el siguiente: si dentro de la estructura del *try* se produce una excepción, se aborta la ejecución actual y se ejecuta el bloque catch cuyo argumento sea compatible con el tipo de objeto lanzado en la excepción. La búsqueda de compatibilidad se realiza sucesivamente sobre los bloques *catch* en el orden en que aparecen en el código hasta que aparece la primera concordancia. Todo ello indica que el orden de colocación de los bloques *catch* es determinante. Por ejemplo, si se incluye un manejador universal (tipo *Exception*), éste debería ser el último.

Si no existe un manejador adecuado a una excepción determinada, se continúa con el lanzamiento de la excepción en la secuencia de métodos invocados.

Si un método puede lanzar una excepción, excepto la clase y subclases de *RuntimeException*, es obligatorio tratarla o indicarlo en la declaración del método. Para ello se añade detrás de los parámetros del método, la sentencia *throws* seguido del tipo de excepción.

En el ejemplo siguiente, el método *m1* no trata la excepción y el método *m2* si la trata.

```
public void m1() throws FileNotFoundException {
   java.io.FileInputStream fis = new java.io.FileInputStream("fichero");
}

public void m2() {
   try {
      java.io.FileInputStream fis = new java.io.FileInputStream("fichero");
   } catch (FileNotFoundException fnfe) {
      // Tratamiento
      fnfe.printStackTrace(); // ?
   }
}
```

Lanzamiento de excepciones

El programador podrá lanzar excepciones de forma explícita. Nos podemos plantear dos causas para provocar excepciones:

- Se lanza un mensaje a nuestro objeto que intenta realizar una acción no permitida, es decir, se invoca un método de nuestro objeto con valores de parámetros incorrectos. En este caso, se lanza una excepción de la clase o subclase de *RuntimeException*. Por ejemplo, en la clase *Natural* se intenta establecer un valor negativo
- Se lanza un mensaje a nuestro objeto que no podemos garantizar que llegue a buen fin. Se debe definir la sentencia *throws* en la definición del método. Por ejemplo, se intenta mandar datos un servidor en Internet que podría no responder, en este caso, queremos asegurarnos que el método que invoca sea consciente que la acción puede fallar, y por lo tanto le obligamos a realizar un *try-catch*

Hay que tener en cuenta que se sobrecarga el sistema y se complica el lanzamiento de mensajes, así que conviene no abusar de esta técnica. Otra forma, con menos carga, sería devolver un *boolean* indicando si la acción ha sido realizada con normalidad.

Presentamos varios ejemplos de lanzamiento de excepciones.

```
public void metodo1() throws Exception {
   if (condicion) throw new Exception("Descripción");
}

public void metodo2() {
   if (condicion) throw new RuntimeException("Descripción");
   if (condicion) throw new NumberFormatException("Descripción");
}
```

Podemos crear nuestras propias excepciones heredando de *Exception* o cualquier subclase.

✍ Implementar la clase Excepciones1. Se lee un valor mediante IO de tipo String y se convierte a int mediante el método Integer.parseInt(...). Se muestra un mensaje indicando como ha ido la acción

✍ Ampliar la clase NumeroNatural, para que provoque una excepción del tipo IllegalArgumentException si se intenta establecer una valor negativo

Tipos genéricos

Los tipos genéricos, también llamados tipos parametrizados, permiten definir un tipo mediante un parámetro, que se le dará valor concreto en la instanciación. Los cambios son realizados por tanto en tiempo de compilación.

Los tipos parametrizados se utilizan especialmente para implementar tipos abstractos de datos: pilas, colas... y otros que permiten almacenar distintos tipos de elementos.

Sin los tipos genéricos, nos vemos a utilizar el tipo *Object*. Teniendo el inconveniente de realizar *casting* para recuperar los datos y perdiendo la capacidad el compilador de realizar un control de tipado.

Para agregar a una clase tipos genéricos, se le añade después del nombre una lista de parámetros que representan tipos. El formato es el siguiente:

```
public class MiClase<tipo1, tipo2...> {...}
```

Aunque podemos utilizar cualquier nombre de parámetros para contener tipos, se recomienda los siguientes nombres descriptivos:

- E. Elemento
- K. Clave
- N. Número
- T. Tipo
- V. Valor
- S,U,V... para 2º, 3º... tipos

Por ejemplo, se define una clase que una dupla con clave y valor (K y V). En el cuerpo de la clase se utilizan los parámetros K y V como si fueran tipos normales, tanto para la declaración de atributos, parámetros de métodos, tipos devueltos de métodos... Algunos ejemplos son:

MiClase.java

```
public class MiClase<K, V> {
   private K atributo1;
   private V atributo2;

   public MiClase(K atributo1, V atributo2) {
      this.atributo1 = atributo1;
```

```java
        this.atributo2 = atributo2;
    }
    public K getAtributo1() {
        return atributo1;
    }
    public V metodo(K parametro1) {
        return this.atributo2;
    }
}
```

Cuando el programador utilice esta clase, deberá establece el tipo de clase para K y el tipo de clase para V, tanto en la declaración, como en la instanciación. Un ejemplo sería:

```java
        MiClase<Integer, String> mc = new MiClase<Integer, String>(0, "");
        String res = mc.metodo(3); // Se aplica boxing para convertir 3 en: new Integer(3)
```

No olvidemos, que también podremos establecer el tipo de clase como *Object*, si quisiéramos que fuese general

A continuación, presentamos un código de ejemplo para una clase llamada Box, que pretende almacenar cualquier instancia, en este caso no se utiliza tipos genéricos:

Box.java

```java
public class Box {
    private Object object;
    public void put(Object object) {
        this.object = object;
    }
    public Object get() {
        return this.object;
    }
}
```

PruebaBox.java

```java
public class PruebaBox {
    public static void main(String[] args) {
        Box intCaja = new Box();
        intCaja.put(new Integer(10));
        Integer unInt = (Integer) intCaja.get();
        System.out.println(unInt);
    }
}
```

En el siguiente ejemplo, se define la misma clase pero mediante tipos genéricos:

BoxG.java

```java
public class BoxG<E> {
    private E elemento;

    public void put(E elemento) {
        this.elemento = elemento;
    }

    public E get() {
        return this.elemento;
    }
}
```

PruebaBoxG.java

```java
public class PruebaBoxG {
    public static void main(String[] args) {
        BoxG<Integer> intCaja = new BoxG<Integer>();
        intCaja.put(new Integer(10));
        Integer unInt = intCaja.get();
        System.out.println(unInt);
    }
}
```

✍ *Cambiar la clase Lista, anteriormente implementada, para que sea genérica*

Enumerados

Los tipos enumerados son un tipo definido por el usuario, que representa un conjunto de constantes. Al ser constantes los valores, se deben escribir en mayúsculas.

Los enumerados se pueden definirse en un fichero independiente (son una clase especial):

Semana.java
```
public enum Semana {
   LUN, MAR, MIE, JUE, VIE, SAB, DOM
}
```

También se pueden definir dentro de una clase, en la zona de atributos, pero no se pueden definir dentro de los métodos:

```
public class Clase{
   public enum Semana { LUN, MAR, MIE, JUE, VIE, SAB, DOM};
   //...
```

Para utilizarlo, se referencia como si fueran constantes estáticas de clase:

```
      Semana dia;
      dia = Semana.MAR;
      System.out.println(dia);
```

Debido a que los tipos enumerados son como una clase de tipo especial en Java, hay muchas acciones que se pueden realizar dentro de un *enum*.

Nos ofrecen dos funciones importantes:

- *valueOf(String s)*, este método te devuelve una referencia de la constante del enumerado que coincide con el String pasado por parámetro
- *values()*, este método te devuelve un array con todos las constantes del enumerado

Mostramos un ejemplo de uso.

```
      Semana dia;
      dia = Semana.valueOf("MAR");
      System.out.println(dia);
      for (Semana una : Semana.values()) {
         System.out.println(una);
      }
```

Un enumerado puede contener constructores, métodos y variables. Veamos un ejemplo:

Planeta.java
```
public enum Planeta {
   MERCURIO(3.303e+23, 2.4397e6),
   VENUS(4.869e+24, 6.0518e6),
   TIERRA(5.976e+24, 6.37814e6),
   MARTE(6.421e+23, 3.3972e6),
   JUPITER(1.9e+27, 7.1492e7),
   SATURNO(5.688e+26, 6.0268e7),
   URANO(8.686e+25, 2.5559e7),
   NEPTUNO(1.024e+26, 2.4746e7),
   PLUTON(1.27e+22, 1.137e6);
   // constante universal de gravedad
   public static final double G = 6.67300E-11;
   public final double masa; // in kilogramos
   public final double radio; // in metros

   private Planeta(double mass, double radius) {
      this.masa = mass;
      this.radio = radius;
   }
   public double gravedadSuperficie() {
```

```
      return G * this.masa / (this.radio * this.radio);
   }
   public double pesoSuperficie(double masa) {
      return masa * this.gravedadSuperficie();
   }
}
```

PruebaEnumerados.java

```
public class PruebaPlanetas {
   public static void main(String[] args) {
      Planeta uno;
      uno = Planeta.TIERRA;
      System.out.println(uno);
      System.out.println(uno.masa + " kg");
      System.out.println(uno.radio + " m");
      System.out.println(Planeta.G);
      System.out.println(uno.gravedadSuperficie());
      System.out.println(uno.pesoSuperficie(100));
   }
}
```

✍ ***Crear el enumerado Meses, con los doce meses y los día de cada uno***

Colecciones de datos

Una problemática que surge frecuentemente en la programación, es organizar una colección de objetos. La primera solución que nos surge es utilizar un array.

```
      String[] nombres = new String[10];
```

Surgen aspectos que se presentan en casi todas las colecciones: ¿cómo gestionar cuando el array se llena?, ¿buscar si un elemento ya existe?, ¿ordenar los elementos?...Cuando el número de objetos crece en la colección, la velocidad de respuesta se vuelve un problema.

Java tiene todo un juego de clases e interfaces para guardar colecciones de objetos, que nos dan soluciones a las problemáticas planteadas. En él, todas las entidades conceptuales están representadas por interfaces, y las clases se usan para proveer implementaciones de esas interfaces.

Collection e Iterator

La interfaz más importante es *Collection<E>*. Una *Collection* es todo aquello que se puede recorrer (o iterar) y de lo que se puede saber el tamaño. Muchas otras clases implementaran *Collection* añadiendo más funcionalidades.

Las operaciones básicas de una *Collection* son:

- *add(T e)*. Añade un elemento
- *clear()*. Borra la colección
- *remove(Object obj)*. Borra un elemento
- *isEmpty()*. Indica si está vacía
- *iterator()*. Devuelve un *Iterator* de la colección, útil para realizar un recorrido de uno en uno.
- *size()*. Devuelve el tamaño de la colección
- *contains(Object e)*. Nos indica si el objeto está contenido. Atención, si el objeto tiene sobrescrito el método *equals* o *hashCode* se utilizará, sino se utilizarán las versiones de la clase *Object*.
- *toArray(...)*. Devuelve una array con el contenido de la colección

Tenemos dos formas de recorrer una colección. Una es con la sentencia *for each*, esta es una buena opción si solo queremos realizar operaciones de lectura. La segunda mediante un *iterator*, resulta mejor cuando queramos realizar modificaciones de la colección.

Las operaciones de un *Iterator* son:

- *hasNext()*. Devuelve true si existen más elementos
- *next()*. Devuelve el siguiente elemento
- *remove()*. Borrar el elemento de la colección

A continuación, mostramos un ejemplo.

```
Collection<String> lista = new ArrayList<String>();
lista.add("uno");
lista.add("");
lista.add("dos");
for (String item : lista)
   System.out.println(item);

Iterator<String> it = lista.iterator();
while (it.hasNext()) {
   String s = it.next();
   if (s.equals("")) it.remove();
}
```

Existen un conjunto de interfaces que heredan de *Collection*, que nos aportan más prestaciones. Algunos de ellos son:

- *List*. Es una colección donde se mantiene la posición de los elementos. No está ordenada y puede haber elementos repetidos
- *Set*. Es una colección en la que no existen elementos repetidos
- *SortedSet*. Además se mantiene un orden.
- *Queue*. Es una colección que mantiene una prioridad de procesamiento

List

Es una lista, no ordenada, en que se mantiene el orden de los elementos, pudiendo acceder a su contenido según la posición. Esta lista crece según las necesidades, con lo que nos podemos olvidar de los tamaños. Este interface hereda de *Collection*, y añade los siguientes métodos:

- *add(int index, E element)*. Se añade en una posición determinada
- *get(int index)*. Se recupera el elemento de una posición
- *set(int index, E element)*. Reemplaza el elemento de una posición

Podemos destacar las siguientes implementaciones del interfaz.

ArrayList

Esta implementación mantiene la lista compactada en un array. Tiene la ventaja de realizar lecturas muy rápidas. El problema está en borrar elementos intermedios, ya que tiene que mover el resto del contenido. Esta clase no está preparada para trabajar con varios hilos; para ello tenemos la implementación *Vector*.

Vector

Es muy parecida al anterior, pero con la diferencia de estar preparada para trabajar con hilos

LinkedList

Esta lista se implementa mediante una lista entrelazada, formada por nodos que apuntan al elemento siguiente y el elemento anterior. Esta implementación favorece las inserciones y el borrado, pero hacen muy lento el recorrido.

Presentamos un ejemplo en Java de la utilización de la colección *List*:

PruebaList.java

```
public class PruebaList {
   public static void main(String[] args) {
      List<String> lista = new ArrayList<String>();
      lista.add("uno");
      lista.add("dos");
```

```
        lista.add("tres");
        lista.add("uno"); // Se permiten elementos repetidos
        IO.out.println(lista);
        lista.remove("dos");
        lista.remove(0);
        IO.out.println("size: " + lista.size());
        IO.out.println("index: " + lista.indexOf("tres"));
        IO.out.println(lista);
    }
}
```

✍ ***Realizar la clase PruebaList que prueba el interface List mediante la implementación de ArrayList***

Set

Es una colección en la que no contiene elementos duplicados. Hereda del interface *Collection*, apenas añade métodos nuevos.

Podemos destacar la implementación HashSet.

HashSet

Esta implementación se basa en una tabla hash. Nos referimos a hash, como una función que genera claves, garantizando que el mismo elemento genera siempre la misma clave. Para obtener la clave, se aplica la función hash al valor devuelto por el método *hashcode* de *Object*. Utilizando esta clave como índice de una array se puede obtener accesos muy rápidos a los objetos. Es posible que dos objetos diferentes generen la misma clave, en ese caso se dice que ha habido una colisión. Este aspecto nos obliga a tener asociado a cada clave más de un elemento. Normalmente, si la capacidad está bien elegida, las colisiones no son muy probables.

No se garantiza que se mantenga el orden a lo largo del tiempo. Pero ofrece un tiempo constante en la ejecución de las operaciones básicas: *add*, *remove*, *contains* y *size*.

Hay que tener cuidado porque en esta implementación se consideran que dos objetos son diferentes si su *hashcode* es diferente, aunque el método *equals* devuelva *true*. Por ello, puede que debamos sobrescribir el método *hashcode* de *Object*.

Presentamos un ejemplo en Java de la utilización de la colección *Set*, nos apoyamos en el interface gráfico de la clase IO:

PruebaSet.java

```
import java.util.HashSet;
import java.util.Set;
import upm.jbb.IO;

public class PruebaSet {
    private Set<String> lista;

    public PruebaSet() {
        lista = new HashSet<String>();
        IO.in.addController(this);
    }

    public void add() {
        IO.out.println(">>> " + lista.add(IO.in.read("add")) + ">>>" + lista.toString());
    }
    public void remove() {
        IO.out.println(">>> " + lista.remove(IO.in.read("remove")) + ">>>" +
                lista.toString());
    }
    public void clear() {
        lista.clear();
        IO.out.println(">>>" + lista.toString());
    }
    public void contains() {
        IO.out.println(">>> " + lista.contains(IO.in.read("contains")));
```

```
    }
    public void size() {
        IO.out.println(">>> " + lista.size());
    }
    public static void main(String[] args) {
        new PruebaSet();
    }
}
```

✍ ***Realizar la clase PruebaSet que prueba el interface Set mediante la implementación de HashSet***

Queue

Esta colección está pensada para organizar una cola (FIFO). Los elementos se añaden por el final, y se extraen por el principio. Dispone de los siguientes métodos:

- *boolean add(E e)*. Inserta un elemento en la cola, provoca una excepción si no existe espacio disponible.
- *E element()*. Recupera sin borrar la cabeza de la cola, si está vacía provoca una excepción
- *boolean offer(E e)*. Inserta, si puede, un elemento en la cola.
- *E peek()*. Recupera sin borrar la cabeza de la cola, si está vacía devuelve null
- *E poll()*. Recupera y borra la cabeza de la cola, o devuelve null si está vacía
- *E remove()*. Recupera y borra la cabeza de la cola, si está vacía provoca una excepción

La clase *LinkedList* también implementa esta interface.

Existe el interface *Deque*, que implementa una cola doble, útil para realizar LIFO. *LinkedList* y *ArrayDeque* implementan este interface.

SortedSet

Esta colección mantiene un orden entre los elementos. Para poder mantener el orden, los objetos deben implementar el interface Comparable, el cual tiene un único método (*compareTo*). Con este método implementamos un orden entre nuestros objetos.

Podemos destacar la siguiente implementación.

TreeSet

Un *TreeSet* mantiene los objetos ordenados en un árbol binario balanceado. Mediante el método *compareTo(),* se establece el orden entre los elementos. Realiza búsquedas relativamente rápidas, aunque más lentas que un *HashSet*. Esta implementación no está preparada para trabajar con hilos.

Map

Un *Map* es una colección de duplas de clave-valor, también conocido como diccionario. Las claves no pueden estar repetidas, si dos elemento tienen la misma clave se considera que es el mismo. *Map* no hereda de *Collection*.

Algunos de sus métodos son:

- *boolean containsKey(Object key)*. Devuelve true si contiene la clave
- *boolean containsValue(Object value)*. Devuelve true si tiene este valor
- *V get(Object key)*. Devuelve el valor asociado a la clave
- *Set<K> keySet()*. Devuelve un Set con las claves del Map
- *V put(K key, V value)*. Asocia un valor a una clave
- *V remove(Object key)*. Elimina un valor
- *Collection<V> values()*. Devuelve una colección con todos los valores

Tiene varias implementaciones, una interesante es *HashMap*.

✍ ***Realizar un ejemplo de utilización de la colección Map***

SortedMap

Es un *Map* ordenado basado en el interface Comparable. Esta ordenado según las claves. Si se utilizan las clases *Integer*, *Double*, *String*... como claves, ya implementan el interface Comparable.

Tiene varias implementaciones, una interesante es *TreeMap*.

Programación concurrente

Principios conceptuales

La plataforma Java nos proporciona las herramientas necesarias para implementar una programación concurrente. Para ello nos aporta la clase *Thread*. A cada parte de un programa concurrente se le llama hilo o hebra (*thread*) y representa un proceso ligero, pudiendo funcionar en paralelo con el resto de hilos. Un proceso ligero, a diferencia de un proceso normal, comparte un mismo espacio de direcciones y requiere menos carga para el cambio de contexto. Un cambio de contexto es pasar el uso de CPU o núcleo de CPU de un proceso a otro.

En la actualidad, la mayoría de los procesadores tiene varios núcleos, por lo que existe una ejecución en paralelo real de los diferentes hilos. Una programación concurrente nos permite acelerar la ejecución de los programas.

Java proporciona mecanismo para la sincronización entre hilos, permitiendo realizar acciones coordinadas.

Cada hilo tiene una prioridad, que establece el orden de ejecución respecto al resto de hilos, cuando estos entran en competición por el uso del núcleo de CPU.

Hilo principal

Cuando un programa Java comienza a ejecutarse, a través de su método *main*, lo realiza a través del hilo principal. Este hilo se crea automáticamente. Podemos obtener una referencia mediante el método estático *currentThread* de la clase *Thread*:

```
Thread hiloPrincipal = Thread.currentThread();
```

A partir de su referencia, podemos modificar las características del hilo principal.

Desde él, se pueden crear nuevos hilos. Nuestra aplicación terminará cuando finalicen todos los hilos de usuario.

Creación de hilos

Para crear nuevos hilos debemos crear instancias de la clase *Thread*, cada instancia representa un hilo diferente. Podemos utilizar dos técnicas diferentes:

- Heredar de la propia clase *Thread* y crear una instancia de la subclase
- Implementando el interface *Runnable* y asociar nuestro objeto a una instancia de *Thread*

Extensión de la clase Thread

Nuestra clase debe heredar de la clase *Thread* y sobrescribir el método *run*. El nuevo hilo ejecutará el cuerpo del método *run*, y finalizará, de forma natural, cuando finalice la ejecución del método. En el constructor de nuestra clase, llamaremos al constructor de la superclase con el nombre del hilo y ejecutaremos el método *start*.

```java
public class HiloH extends Thread {

   public HiloH(String nombre) {
      super(nombre);
      this.start();
   }

   @Override
   public void run() {
      for (int i = 0; i < 10; i++)
         System.out.println("Soy " + this.getName());
   }
}
```

Esta clase se ejecuta de forma concurrente, ya que cuando creamos una nueva instancia se ejecuta en un hilo diferente.

```java
   public static void main(String[] args) {
      new HiloH ("Uno");
      new HiloH ("dos");
      new HiloH ("tres");
   }
```

En este ejemplo, el código que ejecuta el hilo principal finaliza después de crear las tres instancias. Cada instancia se crea en un hilo diferente. La aplicación finaliza cuando se haya terminado la ejecución de todos los hilos. El orden de ejecución es indeterminado.

Implementación del interfaz Runnable

Ésta suele ser la forma más común para la creación de hilos. Nuestra clase debe implementar el interfaz *Runnable*, que tiene un solo método: *run*. Se creará una instancia de *Thread*, asociando la referencia de nuestro objeto.

Hilo.java

```java
public class Hilo implements Runnable {
   private Thread t;

   public Hilo (String nombre) {
      this.t = new Thread(this, nombre);
      t.start();
   }

   @Override
   public void run() {
      for (int i = 0; i < 10; i++)
         System.out.println("Soy " + t.getName());
   }
}
```

```java
   public static void main(String[] args) {
      new Hilo ("Uno");
      new Hilo ("dos");
      new Hilo ("tres");
   }
```

✍ ***Realizar los dos ejemplos anteriores***

FINALIZACIÓN DE HILOS

De forma natural, un hilo finaliza cuando se termina la ejecución del método *run*. Aunque existen métodos para forzar una finalización, no se deben utilizar. Resulta ***desaconsejable*** el uso de *stop*, *destroy*, *resume* y *suspend,* debido a que podemos dejar a toda la aplicación inestable.

En lugar de ello, debemos implementar mediante código, la preparación del hilo para una finalización natural. Mostramos a continuación un posible ejemplo.

Hilo2.java
```
public class Hilo2 implements Runnable {
   private Thread t;
   private boolean stop;

   public Hilo2(String nombre) {
      this.t = new Thread(this, nombre);
      t.start();
   }

   public void setStop(boolean stop) {
      this.stop = stop;
   }
   @Override
   public void run() {
      while (!this.stop) {
         System.out.println("Soy " + t.getName());
      }
   }
   public static void main(String[] args) {
      Hilo2 h2 = new Hilo2("Hilo2");
      h2.setStop(true);
   }
}
```

ATRIBUTOS DE LOS HILOS

Nombre (Name)

Es el nombre asignado al hilo. Para leer o establecer el nombre se realiza con los métodos *getName* y *setName*.

Prioridad (Priority)

El planificador es la parte de la máquina virtual de Java que se encarga de decidir que hilo entra en ejecución. Los hilos disponen del atributo prioridad, que establece la prioridad para que el planificador les asigne tiempo de ejecución de los núcleos de la CPU.

El valor de la prioridad debe estar entre las constantes de *Thread* MAX_PRIORITY (10) y MIN_PRIORITY (1). También se dispone de la constante NORM_PRIORITY (5). Para leer o establecer la prioridad son *setPriority* y *getPriority*.

Demonio (Daemon)

Los hilos demonio se ejecutan con prioridad baja y proporcionan un servicio básico a un programa o programas cuando la actividad de la máquina es reducida. Los hilos demonio son útiles cuando un hilo debe ejecutarse en segundo plano durante largos períodos de tiempo. Un ejemplo de hilo demonio que está ejecutándose continuamente es el recolector de basura (*garbage collector*).

Este atributo se debe establecer antes del arranque (*start*) y se puede leer en cualquier momento, mediante los métodos *setDaemon* o *getDaemon*.

Si los hilos de usuario finalizan, la aplicación finaliza, abortándose todos los hilos demonio de forma automática.

Estado (State)

El hilo puede estar en los siguientes estados:

- NEW. Se ha creado el hilo, pero todavía no se ha iniciado su ejecución (no se ha invocado a *start*)
- RUNNABLE. Este hilo se está ejecutando
- TERMINATED. El hilo ha finalizado su ejecución
- TIMED_WAITING. Este hilo se encuentra en espera durante un tiempo especificado
- WAITING. El hilo se encuentra esperando de forma indefinida por la acción de otro hilo
- BLOCKED. El hilo se encuentra bloqueado por el monitor

Para conocer el estado de un hilo se dispone del método *getState*.

Existen varios métodos que modifican el estado de un hilo. Algunos de ellos son:

- *start()*. Inicia la ejecución de un hilo, se pasa al estado de RUNNABLE
- *sleep()*. Duerme un hilo durante un determinado tiempo, se pasa al estado de TIMED_WAITING

DORMIR HILOS

Se pueden dormir un hilo mediante el uso del método estático *sleep*. El hilo que ejecuta el método, se duerme el tiempo especificado por parámetro en milisegundos.

El hilo que ejecuta *sleep* pasa al estado TIMED_WAITING. Se puede salir del estado por dos razones: una es cuando expira el tiempo, y la otra que otro hilo ha ejecutado el método *interrupt* sobre el actual, produciendo en este caso una excepción.

A continuación mostramos un ejemplo. Observar que al invocar al constructor de *HiloSleep*, es decir, al crear una nueva instancia, se crea un nuevo hilo que se encarga de escribir el mensaje. En este ejemplo, la clase *PruebaHiloSleep* se ejecuta en paralelo con la instancia de *HiloSleep*.

HiloSleep.java

```
public class HiloSleep implements Runnable {
   private Thread t;

   public HiloSleep(String nombre) {
      this.t = new Thread(this, nombre);
      t.start();
   }

   @Override
   public void run() {
      for (int i = 0; i < 10; i++) {
         System.out.println("Soy " + t.getName());
         try {
            Thread.sleep(1000);
         } catch (InterruptedException e) {}
      }
   }
}
```

PruebaHiloSleep.java

```
public class PruebaHiloSleep {
   public static void main(String[] args) {
      new HiloSleep("Hilo2");
      try {
         Thread.sleep(3500);
      } catch (InterruptedException e) {}
      System.out.println("Sigue el hilo principal...");
   }
}
```

✍ ***Realizar la clase CuentaAtras. En el constructor se establece el valor de la cuenta atrás en segundos y funciona mediante un hilo diferente. Por cada segundo muestra un mensaje (por IO) del tiempo que falta. Cuando llegue a cero, lanza un mensaje final (por IO)***

Exclusión y sincronización

EXCLUSIÓN MUTUA

La exclusión mutua garantiza que una porción de código se ejecuta por un sólo hilo, y hasta que el hilo en curso no finalice, no puede entra a ejecutar otro hilo. Si un hilo intenta entrar y está ocupado pasará al estado *BLOCKED*

El lenguaje Java nos permite establecer la ejecución en exclusión mutua a dos niveles: mediante métodos o mediante objetos.

Para forzar a un método que se ejecute mediante exclusión mutua se realiza añadiendo la palabra *synchronized*. Cuando un hilo ejecute el método, bloqueará su entrada para el resto de hilos. Mostramos un ejemplo.

```
    private static int contador;
    public static synchronized void exclusionMutua() {
        contador++;
        contador %= 10;
    }
```

En el siguiente ejemplo, se lanzan 1.000 hilos, y cada uno invoca 10.000 veces al método incContadador. El resultado final debe ser 10.000.000. Cuando quitamos la palabra *synchronized*, existe un mal funcionamiento del programa.

HiloContador.java

```
public class HiloContador implements Runnable {
    private Thread t;
    private static int contador = 0;
    public HiloContador() {
        this.t = new Thread(this, "");
        this.t.start();
    }
    public static synchronized void incContador() {
        HiloContador.contador++;
    }
```

```
    public Thread getT() {
        return this.t;
    }
    public static int getContador() {
        return HiloContador.contador;
    }
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            HiloContador.incContador();
        }
    }
    public static void main(String[] args) {
        for (int i = 0; i < 1000; i++)
            new HiloContador();
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            System.out.println("Me han interrumpido");
        }
        System.out.println(HiloContador.getContador());
    }
}
```

✍ ***Probar el ejemplo anterior con y sin synchronized***

La exclusión mutua de objetos o arrays se realiza con la siguiente estructura gramatical:

```
    synchronized (objeto) {...}
```

Mostramos un ejemplo.

```
        public static int[] array = new int[10];
        //...
        synchronized (array) {
            for (int i = 0; i < array.length; i++)
                array[i] = 0;
        }
```

Se debe tener cuidado ya que Java no controla los interbloqueos. Un interbloqueo ocurre cuando dos o más hilos se quedan en estado bloqueado, con parte de los recursos, y esperando que el resto de hilos libere los recursos que les falta, que a su vez están bloqueados, por los recursos que se están en su poder.

SINCRONIZACIÓN

El lenguaje Java nos proporciona varios mecanismos para la sincronización de hilos.

Mediante el método *join* de la clase *Thread* se espera a que otro hilo muera. Se puede establecer un tiempo máximo de espera. Cuando se utiliza este método (además de *sleep*, *wait*...) se obliga a realizar un *try-catch*, ello es debido a que se puede abortar la interrupción de la espera mediante el método *interrupt*, y en tal caso, se produce la excepción *InterruptedException*.

A continuación, cambiamos el método *main* del ejemplo anterior, para realizar una sincronización.

```
   public static void main(String[] args) {
      Thread[] ts = new Thread[1000];
      for (int i = 0; i < 1000; i++)
         ts[i]=new HiloContador().getT();
      for (int i = 0; i < 1000; i++)
         try {
            ts[i].join();
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      System.out.println(HiloContador.getContador());
   }
```

Mediante los métodos *wait* y *notityAll* heredados de *Object* se pueden sincronizar el modelo productor-consumidor, pero sólo se pueden utilizar dentro de métodos *synchronized*.

En el ejemplo que se muestra, se implementa un ejemplo de este modelo.

Buffer.java

```
package concurrente;
public class Buffer {
   private String[] buffer;
   private int indiceEscritura = 0, indiceLectura = 0, numElemtos = 0;

   public Buffer(int size) {
      this.buffer = new String[size];
   }

   public synchronized void put(String elemento) {
      while (numElemtos == buffer.length)
         try {
            wait();
         } catch (InterruptedException e) {
            System.out.println("Interrupt()");
         }
      buffer[indiceEscritura] = elemento;
      indiceEscritura = (++indiceEscritura) % buffer.length;
      numElemtos++;
      notifyAll();
   }

   public synchronized String get() {
      while (numElemtos == 0)
         try {
            wait();
         } catch (InterruptedException e) {
            System.out.println("Interrupt()");
         }
      String dato = buffer[indiceLectura];
      indiceLectura = (++indiceLectura) % buffer.length;
      numElemtos--;
      notifyAll();
      return dato;
   }
}
```

Consumidor.java

```
package concurrente;
import java.util.Random;

public class Consumidor extends Thread {
```

```java
    private Buffer buffer;
    private boolean stop;

    public Consumidor(Buffer buffer, String nombre) {
        super(nombre);
        this.buffer = buffer;
        this.stop = false;
        this.start();
    }

    public boolean isStop() {
        return stop;
    }

    public void setStop(boolean stop) {
        this.stop = stop;
    }

    @Override
    public void run() {
        Random r = new Random();
        while (!stop) {
            try {
                Thread.sleep(r.nextInt(2000));
            } catch (InterruptedException e) {}
            System.out.println(this.getName() + ":" + buffer.get());
        }
    }
}
```

Productor.java

```java
package concurrente;

import java.util.Random;

public class Productor extends Thread {
    private Buffer buffer;
    private boolean stop;
    private int producto;

    public Productor(Buffer buffer, String nombre) {
        super(nombre);
        this.buffer = buffer;
        this.stop = false;
        this.producto = 0;
        this.start();
    }

    public boolean isStop() {
        return stop;
    }

    public void setStop(boolean stop) {
        this.stop = stop;
    }

    @Override
    public void run() {
        Random r = new Random();
        while (!stop) {
            try {
                Thread.sleep(r.nextInt(2000));
            } catch (InterruptedException e) {}
            buffer.put(this.getName() + "(" + this.producto++ + ")");
        }
    }
}
```

AplicacionBuffer.java

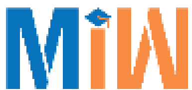
```java
package concurrente;

public class AplicacionBuffer {
    public static void main(String[] args) {
        Buffer buffer = new Buffer(10);
        for (int i = 0; i < 3; i++) {
            new Productor(buffer, "p" + i);
```

```
            new Consumidor(buffer, "c" + i);
        }
    }
}
```

Se debe tener cuidado con las clases que no soportan la multitarea, Java nos proporciona un conjunto de paquetes con clases apropiadas a la multitarea:

- java.util.concurren
- java.util.concurrent.atomic
- java.util.concurrent.locks



3. Flujos de datos

Clases básicas

Clases InputStream y OutputStream

Un flujo de datos (*stream*) representa una fuente datos o un destino. La potencia de estas clases es que trata de igual manera los diferentes canales de comunicación. Un flujo de datos puede provenir o dirigirse hacia archivos en disco, dispositivos de comunicaciones, otros programas o arrays en memoria.

Java basa los flujos de datos mediante dos clases abstractas: *InputStream* y *OutputStream* del paquete *java.io*. Los datos pueden ser bytes, tipos primitivos, caracteres propios de un idioma local u objetos.

Los flujos pueden simplemente transferir datos sin modificación o manipular esos datos para transformarlos de diversas maneras como parte del proceso de transferencia. Sea cual sea el tipo de datos, un flujo representa una secuencia de datos.

Un flujo de entrada (*input stream*) lee datos de una fuente. Un flujo de salida (*output stream*) escribe datos en un destino. Ambas clases, manejan los datos como una secuencia de bytes. Existe un puntero oculto que apunta al byte actual, cada vez que se realice una operación de lectura o escritura, el puntero avanza automáticamente.

La clase *InputStream* es para lectura y cuenta con los siguientes métodos principales:

- *int read().* Lee un *byte* y lo devuelve como *int*. Si no puede leer, devuelve un -1
- *int read(byte[] b).* Lee un conjunto de bytes y los almacena en el array pasado por parámetro. El número de bytes leído lo indica en el entero devuelto. Si no existen bytes disponibles devuelve -1
- *long skip(long n).* Avanza n bytes en el flujo de entrada. Devuelve el número de bytes saltados, el valor es negativo si no salta ninguno.
- *close().* Cierra la conexión y libera los recursos. Esta acción es obligatoria y hay que realizarla aunque se produzcan errores de E/S.

La clase *OutputStream* es para la escritura y dispone de los siguientes métodos principales:

- *write(int b)*. Escribe el byte menos significativo de b
- *write(byte[] b)*. Escribe el array de bytes
- *close()*. Cierra el flujo y libera los recursos

Aunque estas clases trabajan con diferentes canales de comunicación, vamos a realizar los primeros ejemplos concretos con *FileInputStream* y *FileOutputStream* por sencillez. Estas clases trabajan con ficheros.

El siguiente ejemplo duplica un fichero. Al poner sólo los nombres de ficheros, se trabaja en la carpeta por defecto de trabajo.

Observar que existen dos secuencias de códigos importantes. La primera establecemos el flujo de datos asociado a un canal de comunicaciones concreto (un fichero), observar en el API que *FileInputStream* hereda de *InputStream*.

```
        is = new FileInputStream("origen.txt"); // si no existe: excepción
        os = new FileOutputStream("destino.txt"); // si no existe se crea
```

Una vez establecido el flujo, se trabaja independiente al canal, este mismo código sería útil para otros canales. Con el método *read* se lee un byte (valor numérico de 0 a 255), cuando el valor devuelto es -1, significa que se ha finalizado el flujo, en nuestro caso, se ha terminado el fichero; por ello, el bucle *while* finaliza al leer -1.

```
        while ((c = in.read()) != -1) {
           out.write(c);
        }
```

Mostramos el ejemplo completo

BytesStream.java

```
package flujos;
import java.io.*;

public class BytesStream {
   public static void main(String[] args) {
      InputStream is = null;
      OutputStream os = null;
      try {
         is = new FileInputStream("origen.txt"); // si no existe: excepción
         os = new FileOutputStream("destino.txt"); // si no existe se crea
         int c;

         while ((c = is.read()) != -1) {
            os.write(c);
         }
      } catch (FileNotFoundException e) {
         System.out.println("Fichero no encontrado");
      } catch (IOException e) {
         System.out.println("Error de E/S");
      }

      if (is != null) try {
         is.close();
      } catch (IOException e) {}

      if (os != null) try {
         os.close();
      } catch (IOException e) {}
   }
}
```

Se debe garantizar que finalmente cerramos las conexiones. Si no tratamos todas las posibles excepciones, se debería añadir la sentencia *finally* para cerrar las conexiones.

✍ ***Probar el ejemplo anterior: BytesStream***

Clases DataInputStream y DataOutputStream

Para el tratamiento de tipos primitivos, Java dispone de las clases *DataInputStream* y *DataOutputStream*. Ambas heredan de sus correspondientes *InputStream* y *OutputStream*. Nos aportan métodos como: *readBoolean*, *readInt*, *readLong*, *readDouble*... y *writeBoolean*, *writeInt*, *writeLong*, *writeDouble*...

Se debe tener cuidado ya que al escribir los datos se pasan a bytes, y al leerlos se interpretan desde bytes. Por lo tanto, si el tipo leído no corresponde con el tipo escrito no da error, pero el valor puede resultar inesperado.

El constructor de *DataInputStream* necesita por parámetro una instancia compatible con *InputStream*, y en nuestro caso como *FileInputStream* hereda de *InputStream* nos es válido para el ejemplo.

A continuación, mostramos un ejemplo de escritura y lectura.

DataStream.java

```
package flujos;
import java.io.*;
public class DataStream {
   public static void main(String[] args) {
      String fichero = "dataStream.bin";
      DataOutputStream dos = null;
      DataInputStream dis = null;

      // -- Escribe el fichero-----------------------------
      try {
         dos = new DataOutputStream(new FileOutputStream(fichero));
         dos.writeBoolean(true);
         dos.writeInt(1234);
         dos.writeDouble(56.67);
      } catch (FileNotFoundException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
      if (dos != null) try {
         dos.close();
      } catch (IOException e) {}
      // -- Lee el fichero----------------------------
      try {
         dis = new DataInputStream(new FileInputStream(fichero));
         // OJO: se debe leer en el mismo orden escrito
         System.out.println(dis.readBoolean());
         System.out.println(dis.readInt());
         System.out.println(dis.readDouble());
      } catch (FileNotFoundException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
      if (dis != null) try {
         dis.close();
      } catch (IOException e) {}
   }
}
```

✍ ***Probar el ejemplo anterior: DataStream***

Clases ObjectInputStream y ObjectOutputStream

Para el tratamiento de objetos, Java nos proporciona las clases *ObjectInputStream* y *ObjectOutputStream*. Estas clases deserializan y serializan los objetos, es decir, a partir de una secuencia de bytes recuperan el objeto o convierten un objeto en una secuencia de bytes. Para que un objeto se pueda serializar debe implementar el interface *java.io.Serializable*, este interface no tiene métodos.

Es conveniente asegurarse de que la versión del objeto serializado coincide con la versión de la clase de destino, para ello se recomienda añadir una constante de tipo *long* a la clase que establece la versión, llamada normalmente *serialVersionUID*. El valor añadido puede ser la fecha de la última modificación de la clase en formato *long*. Un ejemplo sería:

```
private static final long serialVersionUID = 1257160399327L; // 2.11.09
```

Estas clases también pueden trabajar con los tipos primitivos de Java.

Usr.java

```
package flujos;
import java.io.Serializable;
public class Usr implements Serializable {
   private static final long serialVersionUID = 1257160399327L; // 2.11.09
```

```java
    private String nombre;
    private long movil;
    public Usr(String nombre, long movil) {
        this.nombre = nombre;
        this.movil = movil;
    }
    public String getNombre() {
        return nombre;
    }
    public long getMovil() {
        return movil;
    }
    @Override
    public String toString() {
        return "(" + this.nombre + "," + this.movil + ")";
    }
}
```

ObjectStream.java

```java
package flujos;
import java.io.*;
public class ObjectStream {
    public static void main(String[] args) {
        String fichero = "objectStream.bin";
        ObjectOutputStream oos = null;
        ObjectInputStream ois = null;

        // -- Escribe el fichero-----------------------------
        try {

            oos = new ObjectOutputStream(new FileOutputStream(fichero));

            oos.writeObject(new Usr("uno", 1));
            oos.writeObject(new Usr("dos", 2));
            oos.writeObject(new Usr("tres", 3));

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        if (oos != null) try {
            oos.close();
        } catch (IOException e) {}

        // -- Lee el fichero----------------------------
        try {

            ois = new ObjectInputStream(new FileInputStream(fichero));

            System.out.println(ois.readObject());
            System.out.println(ois.readObject());
            System.out.println(ois.readObject());

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        if (ois != null) try {
            ois.close();
        } catch (IOException e) {}
    }
}
```

✍ ***Probar el ejemplo anterior: ObjectStream***

Clases BufferedInputStream y BufferedOutputStream

En los apartados anteriores, se han realizado las operaciones de entrada y salida, de forma directa. Esto significa que cada petición es manejada directamente por el sistema operativo. Esto puede hacer que el programa sea menos eficiente, ya que cada solicitud de este tipo suele provocar el acceso a disco, la actividad de la red, o alguna otra operación que es relativamente costosa.

Para reducir este tipo de gastos generales, la plataforma Java implementa buffers de E/S. Los datos se lee o escriben sobre una zona de memoria, y sólo cuando queda vacía o se llena se realiza la operación de E/S. Presentamos algunos ejemplos:

```
Is = new BufferedInputStream( new FileInputStream(fichero) );

os = new BufferedOutputStream( new FileOutputStream(fichero) );
```

```
dis =   new DataInputStream(
        new BufferedInputStream(
        new FileInputStream(fichero)));

dos =   new DataOutputStream(
        new BufferedOutputStream(
        new FileOutputStream(fichero)));
```

```
oos =   new ObjectOutputStream(
        new BufferedOutputStream(
        new FileOutputStream(fichero)));

ois =   new ObjectInputStream(
        new BufferedInputStream(
        new FileInputStream(fichero)));
```

El resto de código sería igual que en los ejemplos mostrados.

Clases InputStreamReader y OutputStreamWriter

Estas clases están preparadas para los flujos de caracteres. La plataforma Java almacena los caracteres utilizando la convención Unicode. En las operaciones de E/S traduce automáticamente al formato internacional desde el formato local o viceversa.

Estas clases, añaden métodos nuevos que permiten trabajar con String: *write (String s)*.

Para un flujo con ficheros, se utilizan las clases *FileReader* y *FileWriter*.

CharsStream.java

```
package flujos;
import java.io.*;
public class CharsStream {
   public static void main(String[] args) {
      InputStreamReader in = null;
      OutputStreamWriter out = null;
      try {
         in = new FileReader("origen.txt"); // si no existe: excepción
         out = new FileWriter("destino.txt"); // si no existe se crea
         int c;

         while ((c = in.read()) != -1) {
            out.write(c);
         }

      } catch (FileNotFoundException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }

      if (in != null) try {
         in.close();
```

```
      } catch (IOException e) {}

      if (out != null) try {
         out.close();
      } catch (IOException e) {}

   }
}
```

Si se utiliza un buffer quedaría:

```
   ...
      BufferedReader br = null;
      BufferedWriter bf = null;
      try {
         br = new BufferedReader( new FileReader("origen.txt"));
         bf = new BufferedWriter(new FileWriter("destino.txt"));
   ...
```

Las clase *BufferedReader* nos aporta un método para leer una línea completa: *readLine* que devuelve un String, y la clase *BufferedWriter* un método para escribir un salto de línea: *newLine*.

✍ Realizar la clase Almacenar, que lea un String (mediante la clase IO) y la escriba en un fichero nuevo utilizando la clase BufferedWriter. Editar el fichero creado con el Notepad++ para ver su contenido

Ficheros y Directorios

Clases FileInputStream y FileOutputStream

Estas dos clases ya la hemos utilizado en los ejercicios anteriores. Su funcionalidad es proporcionarnos un *InputStream* u *OutputStream* asociado con un fichero. Dispone de dos constructores fundamentales, uno se construye a partir de un String que representa una ruta de un fichero, y otro, se construye a partir de una instancia de la clase *File*.

Clase File

La clase *File* nos facilita la generación de código para examinar y manipular ficheros y carpetas, independientemente de la plataforma. El nombre de esta clase puede dar a engaño, representa nombres de archivos o nombres de carpetas.

Mostramos algunos de sus métodos fundamentales:

- *isDirectory()* o *isFile()*. Indica si representa un directorio o un fichero
- *getPath()*. Ruta almacenada
- *getName()*. Nombre del directorio o fichero
- *getCanonicalPath()* o *getCanonicalFile()*. Ruta completa
- *listRoots()*. Devuelve una lista con las raíces
- *list()* o *listFiles()*. El contenido del directorio
- *getParent()* o *getParentFile()*. Devuelve la ruta del padre o null
- *listFiles(FileFilter ff)*. Lista filtrada por las extensiones. El filtro se debe implementar por el usuario mediante el interface *FileFilter*, tiene un único método llamado *accept(File f)*, hay que devolver true o false si el fichero cumple los requisitos del filtro
- *mkdir()*. Crea un directorio

Podemos crear una instancia de *File* asociada a un fichero:

```
      File fich = new File("origen.txt");
```

Se buscará el fichero en la carpeta actual de trabajo. Si queremos saber la ruta absoluta, podríamos utilizar el método *getCanonicalFile*:

```
      try {
         System.out.println("CanonicalFile: " + fich.getCanonicalFile());
      } catch (IOException e) {}
```

También podemos crear una instancia de *File*, asociado a la carpeta actual de trabajo:

```
      File carpeta = new File(".");
```

Si quisiéramos ver todo el contenido completo de la carpeta, se podría ejecutar el siguiente código:

```
for (File file : carpeta.listFiles())
         System.out.println("   " + file.getName());
```

Para ver todas las raíces instaladas en nuestro equipo, se realizaría con el siguiente código:

```
      for (File raiz : File.listRoots())
         System.out.println("   " + raiz.getPath());
```

A continuación, mostramos un ejemplo completo:

PruebaFile.java

```
package flujos;

import java.io.File;
import java.io.FileFilter;
import java.io.IOException;

public class PruebaFile implements FileFilter {
   // Filtro
   public boolean accept(File f) {
      return f.getName().endsWith(".java");
   }

   public static void main(String args[]) {
      PruebaFile filtro = new PruebaFile();

      // Se asocia a un fichero
      File fich = new File("origen.txt");
      System.out.println("Name: " + fich.getName());
      try {
         System.out.println("CanonicalFile: " + fich.getCanonicalFile());
      } catch (IOException e) {}

      // Se obtiene la ruta actual
      File carpeta = new File(".");
      try {
         carpeta = carpeta.getCanonicalFile();
      } catch (IOException e) {
         System.out.println(e);
      }

      System.out.println("Name:" + carpeta.getName());
      System.out.println("Path: " + carpeta.getPath());
      System.out.println("Parent: " + carpeta.getParent());
      System.out.println("File: " + carpeta.isFile());
      System.out.println("Directory: " + carpeta.isDirectory());

      // Se lista las raices
      System.out.println("Raices: ");
      for (File raiz : File.listRoots())
         System.out.println("   " + raiz.getPath());

      // Se lista el contenido de la carpeta
      System.out.println("Contenido de carpeta actual: ");
      for (File file : carpeta.listFiles())
         System.out.println("   " + file.getName());

      // Se lista los ficheros *.java
      System.out.println("Contenido de carpeta actual: ");
      for (File file : carpeta.listFiles(filtro))
         System.out.println("   " + file.getName());

      // Separador del S.O. actual
      System.out.println("Separador: " + File.separator);
   }
}
```

✍ *Probar el ejemplo anterior*

RandomAccessFile

Mediante la clase *RandomAccesFile*, la plataforma Java nos proporciona el manejo de un fichero de una forma directa, es decir, leyendo o escribiendo en lugares concretos del fichero.

Un archivo de acceso aleatorio se comporta como un gran conjunto de bytes almacenados en el sistema de archivos. Hay un cursor, o el índice en la matriz implícita, llamado puntero de archivo. Las operaciones de entrada o salida se realizan a partir del puntero.

Un archivo se puede abrir como sólo lectura, o lectura y escritura.

La clase dispone de los siguientes métodos fundamentales:

- *Constructores*: *(String fich, String modo)* o *(File fich, String modo)*. El modo puede ser "r" para solo lectura y "rw" para lectura y escritura.
- *length*. Tamaño del fichero
- *read()*, *read(byte[] b)*, *readBoolean()*, *readChar()*…
- *write(int b)*, *write(byte[] b)*, *writeBoolean*, *writeChar()*…
- *readLine()*, *writeChars(String s)*
- *seek(long pos)*, *setLength()*. sitúa el puntero en una posición concreta

PruebaRandom.java

```
package flujos;
import java.io.IOException;
import java.io.RandomAccessFile;
public class PruebaRandom {
   public static void main(String[] args) {
      final int UNIDAD = Integer.SIZE / 8;
      RandomAccessFile raf;
      try {
         raf = new RandomAccessFile("ficheroAleatorio.bin", "rw");
         for (int i = 0; i < 20; i++)
            raf.writeInt(i);
         raf.close();
      } catch (IOException e) {
         System.out.println("Error de E/S: " + e.getMessage());
      }

      try {
         raf = new RandomAccessFile("ficheroAleatorio.bin", "rw");
         raf.seek(5 * UNIDAD);
         int valor = raf.readInt();
         System.out.println(valor);
         raf.seek(5 * UNIDAD);
         raf.writeInt(valor * 10);
         raf.seek(5 * UNIDAD);
         System.out.println(raf.readInt());
         raf.close();
      } catch (IOException e) {
         System.out.println("Error de E/S: " + e.getMessage());
      }
   }
}
```

FileDialog

Aunque esta clase pertenece al paquete *awt*, presentamos un pequeño ejemplo para facilitar la búsqueda de ficheros. Con *getDirectory* te devuelve un String con el directorio, y con *getFile* devuelve un String con el nombre del fichero.

PruebaFileDialog.java

```
package flujos;
import java.awt.FileDialog;
public class PruebaFileDialog {
   public static void main(String args[]) {
```

```
        FileDialog fd = null;
        fd = new FileDialog(fd, "Abrir", FileDialog.LOAD); // FileDialog.SAVE
        fd.setVisible(true);
        fd.dispose();
        System.out.println("getDirectory: " + fd.getDirectory());
        System.out.println("getFile: " + fd.getFile());
    }

}
```

URL

Una URL (*Uniform Resource Locator*) es una referencia a un recurso de Internet. Tiene el siguiente formato:

```
Protocolo://{servidor|alias}.dominio[:puerto][rutaLocal][#marca]
```

Algunos ejemplos de referencias son:

```
http://localhost:8080
http://138.100.152.168/carpeta/pagina.html
http://www.google.com
```

La plataforma Java nos proporciona la clase URL. Con esta clase podemos acceder a los recursos de Internet. La forma más fácil de crear una instancia es a partir de un String que represente una URL. Un ejemplo sería:

```
 unaURL = new URL("http://www.upm.es");
```

La clase URL dispone de un conjunto de métodos que nos devuelven características del recurso de Internet. Algunos de los métodos son:

- *getDefaultPort*. Obtiene el número de puerto por defecto del protocolo asociado
- *GetFile*. Obtiene el nombre del archivo
- *GetHost*. Obtiene el nombre de host de la presente URL
- *getPath*. Obtiene la ruta
- *getPort*. Obtiene el número de puerto
- *getProtocol*. Obtiene el nombre de protocolo
- *getQuery*. Obtiene la parte de consulta
- *getRef*. Obtiene la referencia de la marca dentro de la página (#marca)
- *openStream*. Devuelve un *InputStream* asociado al recurso

A continuación, mostramos un ejemplo de los métodos referenciados.

```
         URL unaURL = new URL("http://www.google.es/search?hl=es&q=url");
        System.out.println("Puerto por defecto: " + unaURL.getDefaultPort());
        System.out.println("File: " + unaURL.getFile());
        System.out.println("Host: " + unaURL.getHost());
        System.out.println("Path: " + unaURL.getPath());
        System.out.println("Protocolo: " + unaURL.getProtocol());
        System.out.println("Puerto: " + unaURL.getPort());
        System.out.println("Consulta: " + unaURL.getQuery());
```

La ejecución del código anterior da como resultado:

```
Puerto por defecto: 80
File: /search?hl=es&q=url
Host: www.google.es
Path: /search
Protocolo: http
Puerto: -1
Consulta: hl=es&q=url
```

Presentamos un programa que lee el contenido de un recurso Web.

UrlStream.java

```
package flujos;
```

```java
import java.io.*;
import java.net.URL;
public class UrlStream {
   public static void main(String[] args) {
      URL unaURL;
      BufferedReader in;
      char[] msgByte = new char[256];
      String msg;
      int longitud;

      try {
         unaURL = new URL("http://www.upm.es");
         in = new BufferedReader(new InputStreamReader(unaURL.openStream()));
         msg = "";
         while ((longitud = in.read(msgByte)) != -1)
            msg += new String(msgByte, 0, longitud);
         System.out.println(msg);
      } catch (IOException e) {
         System.out.println("Error:" + e);
      }
   }
}
```

✍ ***Realizar una clase que pide una url (mediante IO) y guarda en un fichero de tipo txt, el contenido de la misma***