

Lenguaje java

Copyright (c) 2007, Jose Daniel Gutiérrez Porset

josedaniel.gutierrez@ehu.es

Actualización de Gaizka Abaroa (gaizka.abaroa@ehu.es) en 2014

Licencia de Uso

Acerca de este documento

Copyright (c) 2007, Jose Daniel Gutiérrez Porset
josedaniel.gutierrez@ehu.es
y 2013, Gaizka Abaroa (gaizka.abaroa@ehu.es)

Se otorga permiso para copiar, distribuir y/o modificar este documento bajo los términos de la Licencia de Documentación Libre de GNU en su versión 1.2 o cualquier otra versión posterior publicada por la Free Software Foundation, siendo todo él invariante. Una copia de la licencia está disponible en la web de la Free Software Foundation, dentro de la sección titulada GNU Free Documentation License.

Este documento ha sido realizado íntegramente con software libre.

Índice

- Licencia de uso de este documento
- Introducción a la programación orientada a objetos
- Introducción a la tecnología java
- Sintaxis básica
- Funciones básicas de E/S
- Clases y Objetos
- Herencia
- Interfaces
- Arrays y Strings
- Paquetes
- Excepciones

Introducción a la programación orientada a objetos

OPP (POO)

Introducción a la OOP

- Formas de programación:
 - **Procedural:** divide cada tarea de un programa en un conjunto de estructuras de datos y funciones
 - **Orientada a Objetos (OOP):** divide cada tarea de un programa en un conjunto de objetos que tienen asociados atributos y métodos
- Los lenguajes pueden ser mezcla de ambas técnicas.
Ejemplos de lenguajes que tienen OOP:

smalltalk, C++, java, python, ruby, C#, php5

Introducción a la OOP

Conceptos

- **Clase:** abstracción de las entidades de la vida real, incluyendo:
 - **Atributos** o características de la clase
 - **Métodos** u operaciones realizables sobre la clase.(Miembros = atributos + métodos)
- **Objeto:** instancia concreta de una clase
- **Mensaje:** lo que se intercambia entre objetos

Ejemplo:

- Programa de control biblioteca: Clase->**foto** Objeto->**cada foto**
- Atributos de foto: tamaño, autor, fecha, cámara, ...
- Métodos de foto: cambiar tamaño, cambiar color, enviar, ...

Introducción a la OOP

- **Clase CWindow:**

- Datos:

- Tamaño
 - Posición
 - Color
 - Barras de desplazamiento
 - Título, ...

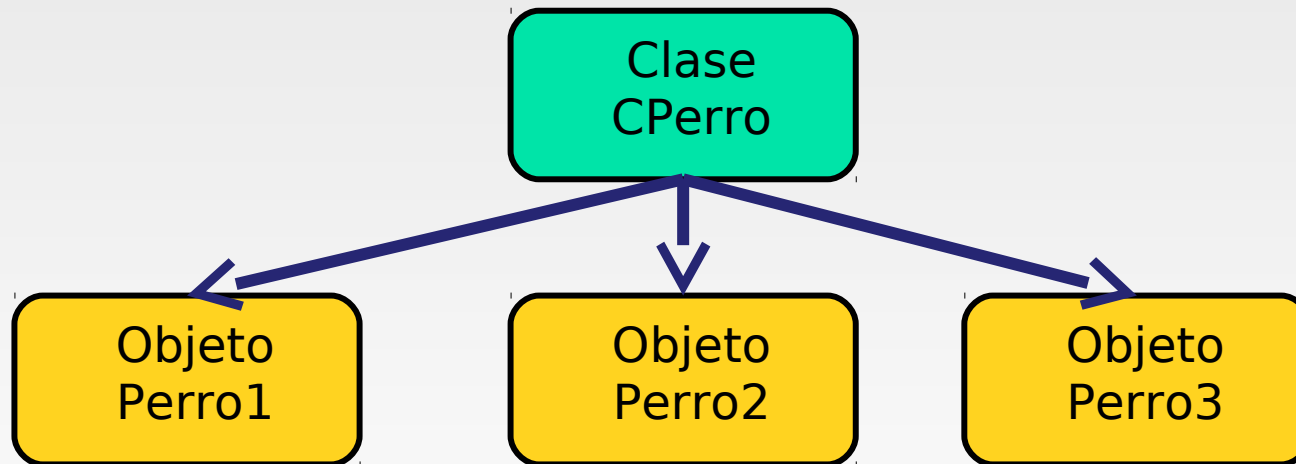
- Métodos:

- Un click de ratón en un botón
 - Maximizar
 - Minimizar
 - Repintar, ...



Introducción a la OOP – Clases y objetos

- Una clase se puede asemejar a una “plantilla” que define datos e incorpora métodos que tienen acceso a esos datos.
- En el momento en el que se declara una clase se incorpora ese tipo de datos al conjunto de tipos de datos disponibles (int, float, ...).
- Un objeto es una variable mapeada en memoria de un tipo clase determinada.



Se crean los objetos Perro1, Perro2 y Perro3 a partir de la clase CPerro

Cada perro tiene sus propios datos

Introducción a la OOP

Conceptos

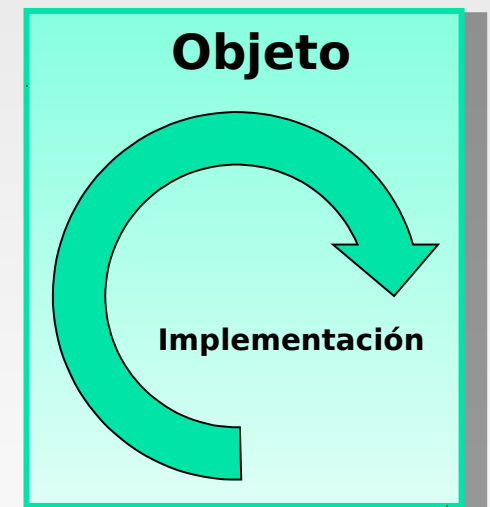
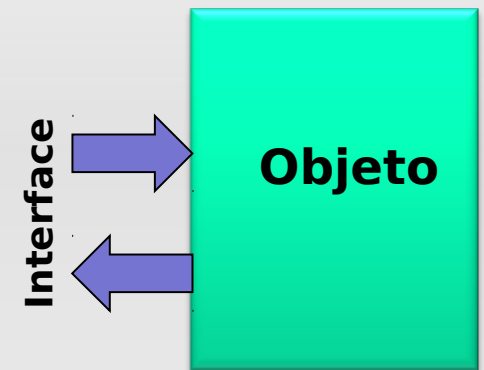
- **Abstracción:** permite hablar de conceptos genéricos: sumar números o sumar cadenas de caracteres.
- **Encapsulamiento:** ocultación de código y datos de objetos (no interesa mostrar todo → simplificar el uso)
- **Herencia:** posibilidad de que una clase adquiera atributos y/o métodos de otra ya existente, pudiendo además añadirse otros miembros, y re-definirse métodos heredados.
A la clase madre se le llama “**superclase**” y a la hija “**subclase**”.

Se crea una **jerarquía de clases**

- **Polimorfismo:** métodos con igual nombre implementan distintas funcionalidades según las clases sobre las que se apliquen. Concepto relacionado: sobrecarga

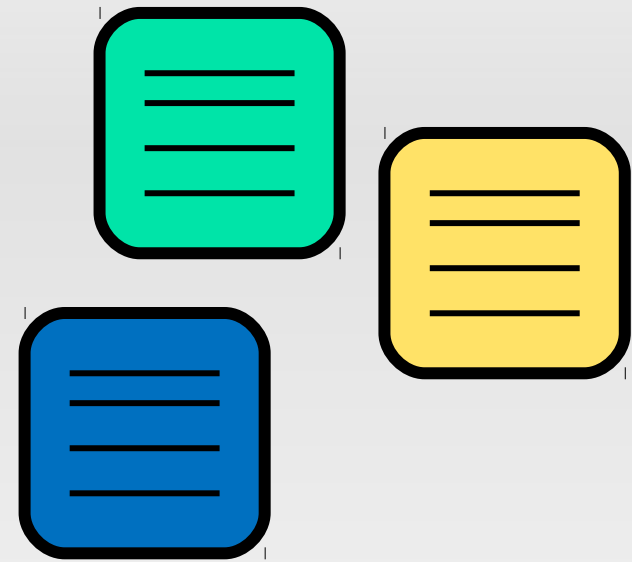
Introducción a la OOP - Abstracción

- **Un concepto** se representa mediante una clase (imagen, rectángulo, libro, biblioteca, ...)
- Los datos se encuentran en **clases**. No hay datos “sueltos”
- Se pueden **ocultar los detalles** de una clase que no interesan al exterior. Desde el punto de vista de la abstracción se distinguen dos aspectos de una clase:
 - Interface: es la parte que un objeto **muestra** o deja **accesible**
 - Implementación: recoge el **cómo** se hace internamente los que esa clase tiene que hacer.
- En resumen: ***nos interesa lo que nos ofrece una clase pero no nos importa cómo lo hace.***



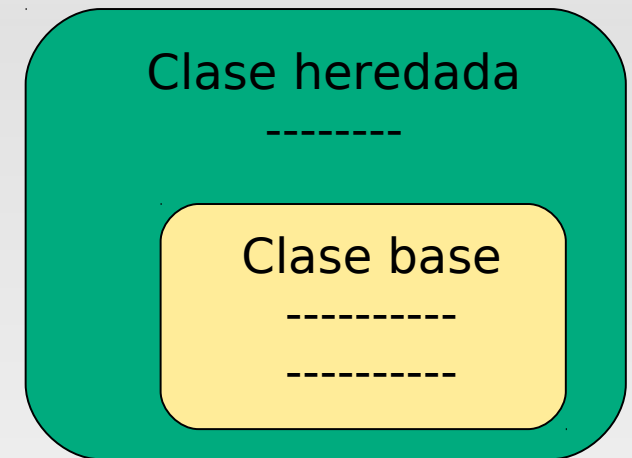
Introducción a la OOP - Encapsulación

- Los datos se encuentran “dentro” de clases. No hay datos “sueltos”
- Comparando con las estructuras se puede controlar el acceso:
 - Datos publicos
 - Datos privados
 - Datos protegidos
- Las funciones tienen relación con los datos. Se aplican a los datos de la clase.
- Aporta “fiabilidad” y “portabilidad” al código



Introducción a la OOP - Herencia

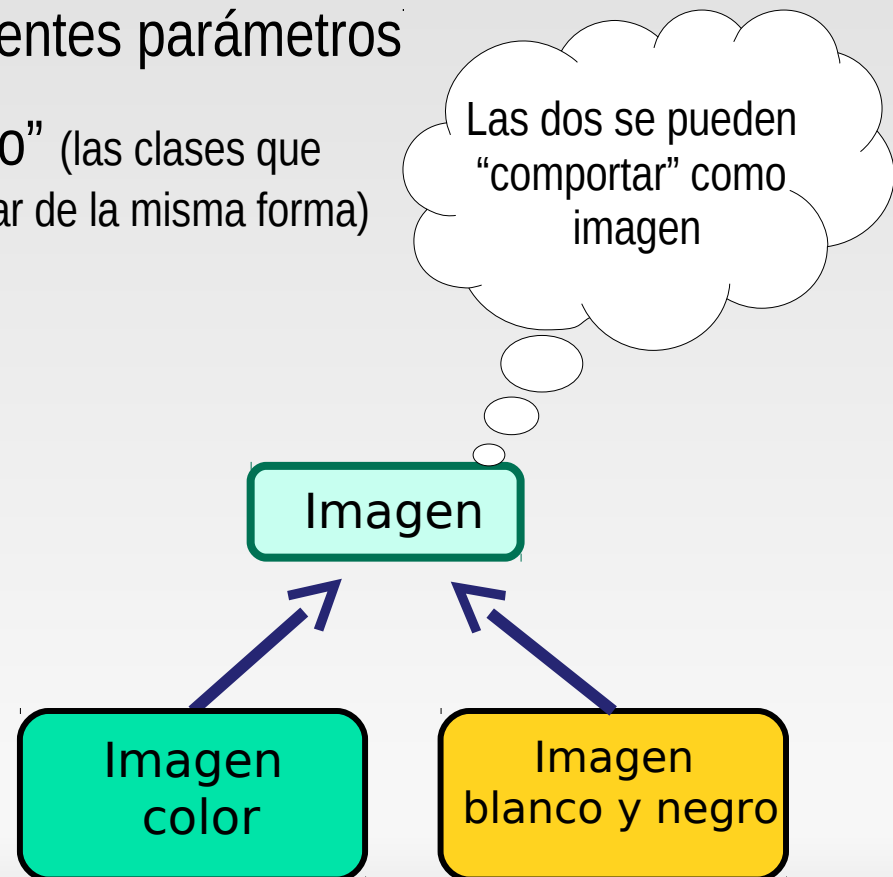
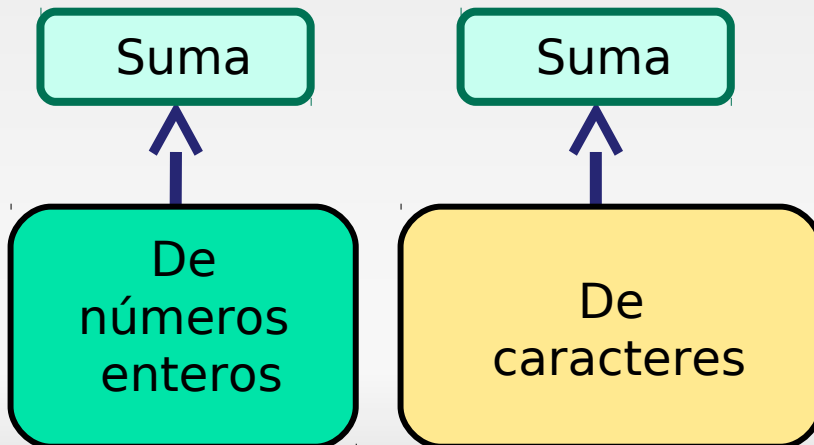
- Gracias al mecanismo de la herencia se puede crear una clase nueva que **“incorpora”** una **clase base** más posibles **“añadidos”**.
- La nueva clase heredada es más **“compleja”** que la base: incorpora nuevos elementos
- Se **“reutiliza”** el código
- Se crea una **jerarquía de clases**
- Se puede dar **herencia múltiple**:
 - una clase puede heredar de varias
 - una clase heredada puede ser base de otra nueva



Importante: la clase heredada “es” también clase base.
Por ejemplo si tenemos la clase CPerro y la clase CDoberman heredada de CPerro, la clase CDoberman **es** a su vez de la clase CPerro.

Introducción a la OOP - Polimorfismo

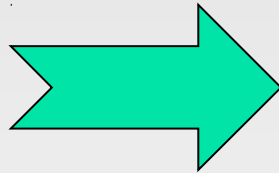
- Elementos diferentes con el mismo “aspecto” (interface)
- Diferentes aspectos del polimorfismo:
 - Funciones con el mismo nombre y diferentes parámetros
 - Clases diferentes con el mismo “aspecto” (las clases que comparten una base vía herencia se pueden comportar de la misma forma)
- Se genera una “**simplicación conceptual**”. Hay menos conceptos diferentes.



OOP Conclusiones

POO (OOP)

- Abstracción
- Encapsulamiento
- Herencia
- Polimorfismo



- Simplificación
- Código más fiable
- Reutilización del código
- Permite abordar proyectos complejos
- *A pesar de cierta sobrecarga aparente*

OOP Conclusiones

- Replanteamiento del proceso de codificación y diseño:
 - *Diseño de clases creando una jerarquía*
 - *Determinar la relación entre clases*
 - *Diseño del interface de cada clase*
 - *Codificación individual de las clases (cada una “en su mundo”)*
 - *Determinación de objetos necesarios*

00P Conclusiones

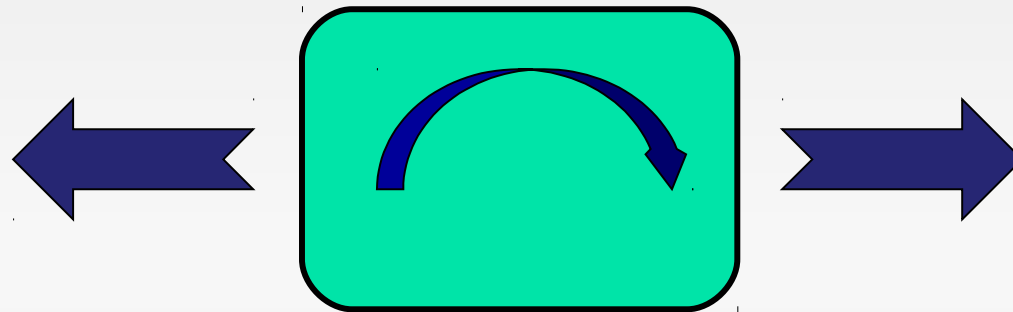
Dos puntos de vista:

1- Codificador de clases:

- Ofrece un **servicio** a los usuarios de esa clase

2- Usuario de la clase:

- No es necesario conocer los detalles sólo el interface





Introducción a la tecnología **Java**

Introducción a la tecnología java

¿Qué es Java?

- Es mucho más que un lenguaje:

Java es un lenguaje de programación y la primera plataforma informática creada por Sun Microsystems en 1995. Es la tecnología subyacente que permite el uso de programas punteros, como herramientas, juegos y aplicaciones de negocios. Java se ejecuta en más de 850 millones de ordenadores personales de todo el mundo y en miles de millones de dispositivos, como dispositivos móviles y aparatos de televisión.

(http://www.java.com/es/download/faq/whatis_java.xml)

Hasta la fecha, la plataforma Java ha atraído a más de 9 millones de desarrolladores de software. Se utiliza en los principales sectores industriales y está presente en una amplia gama de dispositivos, computadoras y redes.

La versatilidad, eficacia, portabilidad de plataformas y seguridad de la tecnología Java la convierte en la tecnología ideal para la informática de redes. Desde portátiles hasta centros de datos, desde consolas para juegos hasta súper computadoras, desde teléfonos móviles hasta Internet, Java está en todas partes.

- 1,100 millones de escritorios ejecutan Oracle Java
- 930 millones de descargas de Java Runtime Environment cada año
- 3 mil millones de teléfonos móviles ejecutan Java
- Se entregan 31 veces más al año teléfonos Java que Apple y Android juntos
- El 100% de los reproductores de Blu-ray ejecutan Java
- Se fabrican 1400 millones de tarjetas Java cada año
- Decodificadores, impresoras, cámaras web, juegos, sistemas de navegación en vehículos, terminales de loterías, dispositivos médicos, estaciones de pago de aparcamientos y mucho más.

Para ver los lugares en los que actúa Java en su vida diaria, [explore java.com](http://www.java.com/es).

- Especificación completa del lenguaje:

<http://java.sun.com/docs/books/jls/>

- Más información en:

<http://java.sun.com/j2se/1.5.0/docs/guide/language/>

http://en.wikipedia.org/wiki/Java_%28programming_language%29

http://en.wikipedia.org/wiki/Java_version_history

<http://www.oracle.com/technetwork/java/index.html>www.java.com/es



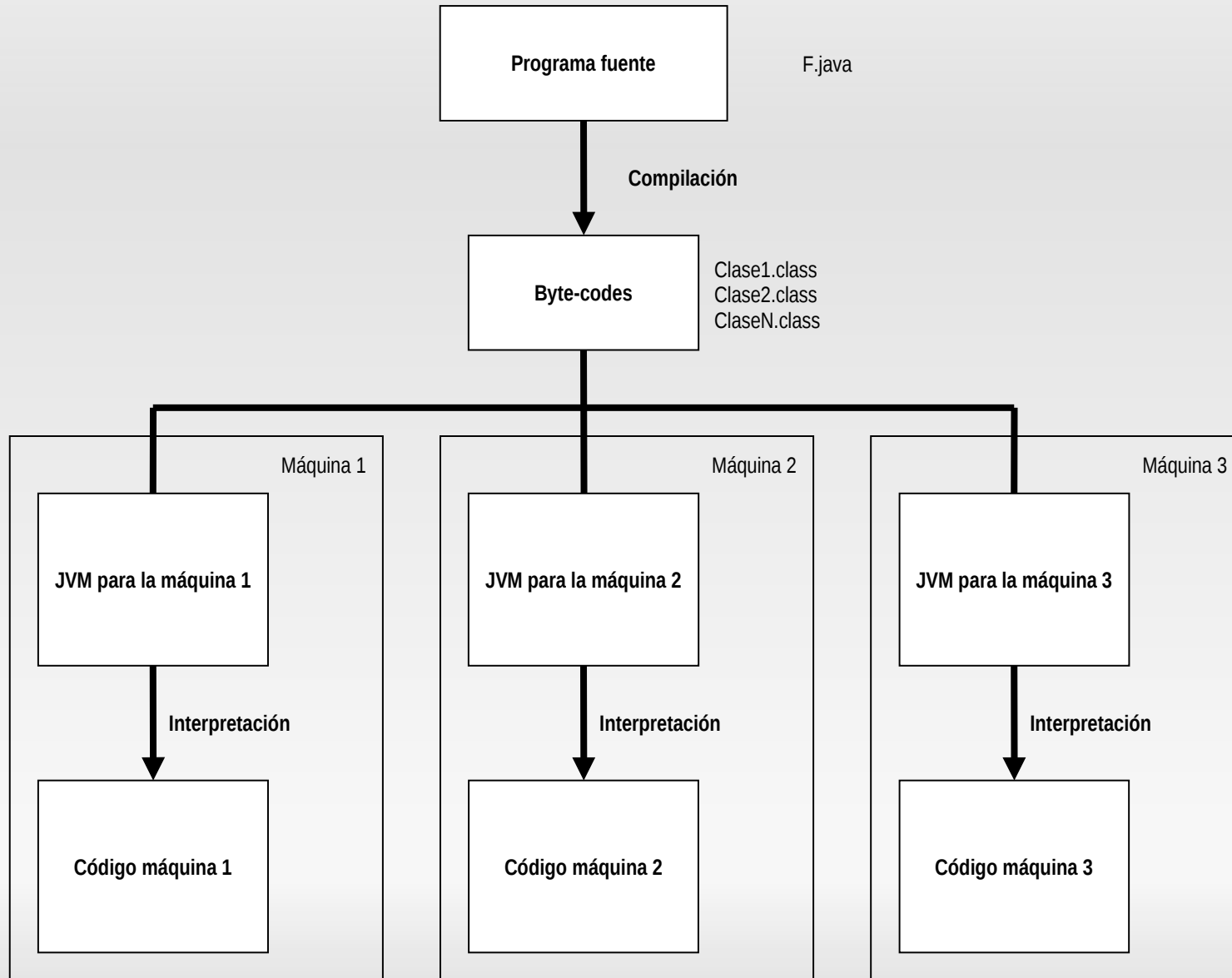
Introducción a la tecnología java

Características generales del lenguaje

- Orientado a objetos
- Lenguaje compilado (bytecodes) e interpretado (JVM).
 - JVM: Java Virtual Machine: Es el intérprete de bytecodes a código binario nativo de cada plataforma (hardware+ sistema operativo).
 - La JVM está implementada en bastantes plataformas HW y de S.O. y se habla de independencia de plataforma. Ej: teléfonos móviles
 - A pesar de ello, también hay:
 - Compiladores puros para java que traducen el código fuente a código máquina nativo.
 - Plugins JIT (Just In Time) para la JVM que aceleran la ejecución en la JVM mediante la compilación de bytecodes a código máquina nativo.
- Proporciona herramientas y librerías para desarrollar código distribuido (ej. RMI)

Introducción a la tecnología java

JVM



Introducción a la tecnología java

JVM

- Gestión de la memoria:
 - El programador no ha de liberar memoria. Esto lo hace la JVM mediante el recolector de basura (“garbage collector”), lanzando un thread de forma automática.
 - No hay punteros. Más seguro que C.
- Gestión del código en ejecución:
 - Carga dinámica y distribuida: distintas partes de un programa se van cargando según se van necesitando, y pueden estar en máquinas diferentes.
 - Ejecución de hilos paralelos (“threads”) con sincronización. Mejor rendimiento para los procesadores de hoy en día
- Acceso a hardware restringido (sólo mediante métodos nativos de otros lenguajes, no utilizables por applets)

Introducción a la tecnología java

JVM

- Seguridad:
 - Comprobación de punteros y de límites de arrays
 - Excepciones
 - Verificación de byte-codes
 - No desbordamientos de pila
 - Parámetros conocidos y correctos
 - No conversiones ilegales de tipos
 - Accesos legales a atributos/métodos de objetos
 - No intentos de violar normas acceso/seguridad
 - Cargador de clases:
 - Separa el espacio de nombres local de los recursos procedentes de la red
 - No caballos de Troya. Las clases se buscan primero entre las locales y luego en exteriores
 - Apertura de ficheros y arranque de otras aplicaciones en máquina 22 remota, no en local

Introducción a la tecnología java

JVM

- Distinguir:
 - Especificación
 - Implementación. Ej. Sun HotSpot (1999) escrita en C++
- Más info en:
 - http://en.wikipedia.org/wiki/Java_Virtual_Machine
 - http://en.wikipedia.org/wiki/List_of_Java_virtual_machines
 - <http://en.wikipedia.org/wiki/HotSpot>

Introducción a la tecnología java

Comparación con C

- Algunas diferencias con C:
 - No tiene preprocesado ni ficheros de cabecera
 - No hay estructuras
 - Gestión de memoria automática → no hay que liberar la memoria (“garbage collector”).
 - No hay punteros.
 - No es posible acceder a elementos fuera de rango (strings y array)
- En general:
 - Más seguro
 - Más lento por ser intepretado, aunque hay compiladores JIT (pierde portabilidad)

Introducción a la tecnología java

Aplicaciones

- Programas “standalone” convencionales. Ej.
 - Software para teléfonos móviles
 - Software para PC (Windows o GNU/Linux), ej. azureus
 - Software para servidor, ej. páginas web dinámicas generadas por un servidor que ejecute servlets o jsp
- Applets: programas ejecutados en un navegador web que tenga cargado un plugin
- Aplicaciones de tipo “Java Web Start”, descargadas a través de internet sin necesidad de un navegador.

Introducción a la tecnología java

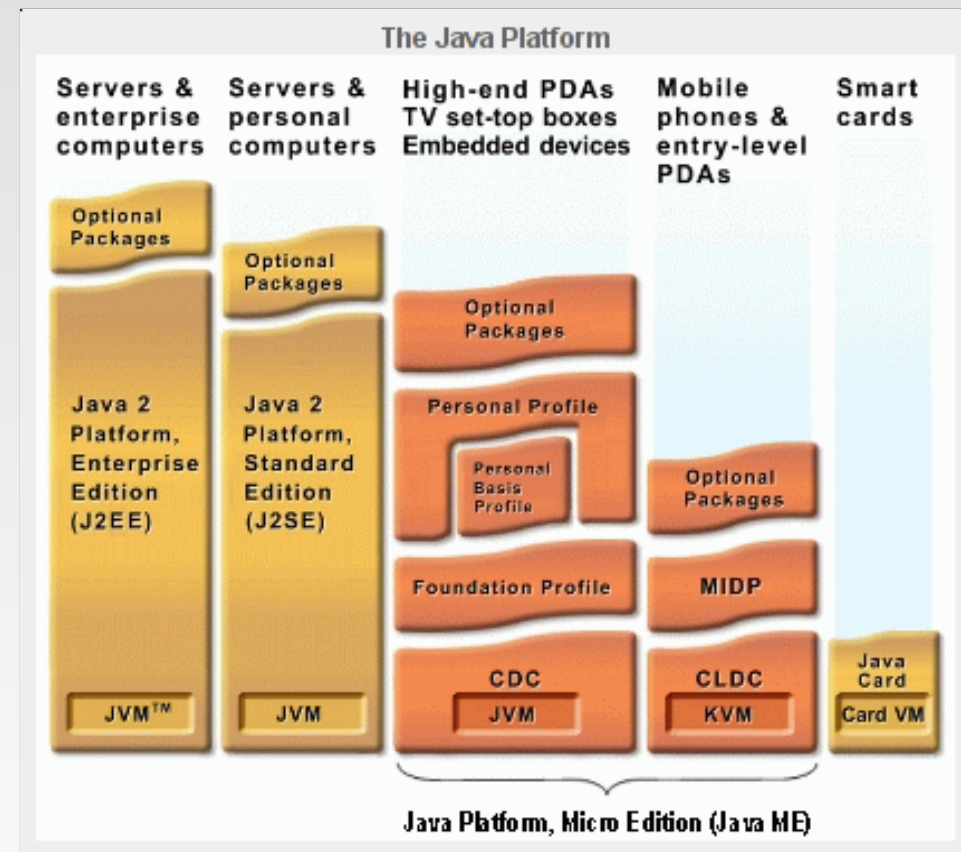
Interfaces gráficos de usuario

- Principales:
 - AWT (de Sun) es el más antiguo
 - Swing (de Sun)
 - SWT (de IBM Eclipse). Apariencia de la plataforma sobre la que corre.
- Bindings para entornos gráficos específicos, ej. KDE/qt, GNOME/gtk,...

Introducción a la tecnología java

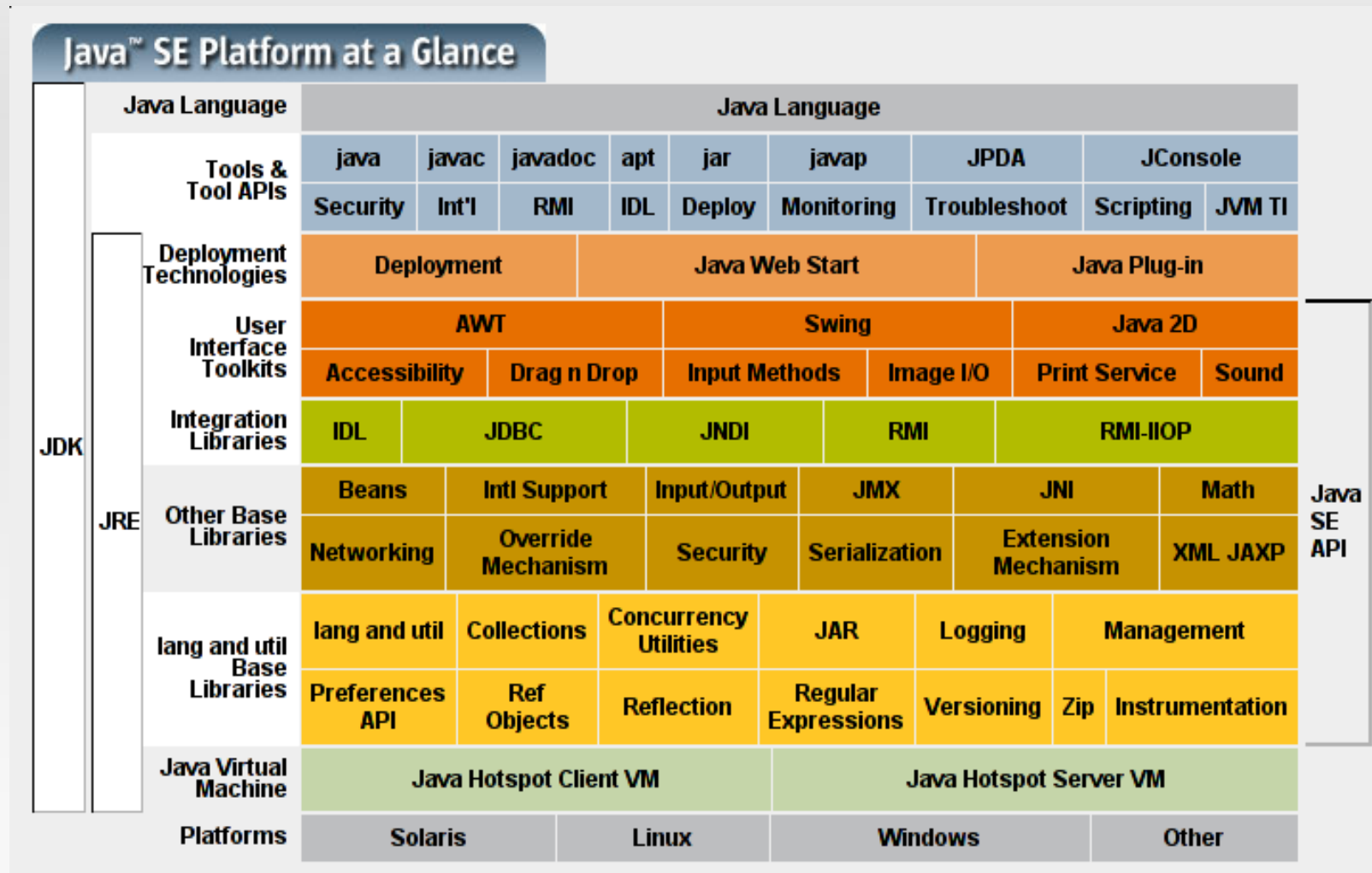
Tecnologías Java de Sun

- Sun tiene distintas familias de Tecnologías:
 - Java Platform, Micro Edition (Java ME): dispositivos ej. móviles
 - Java Platform, Standard Edition (Java SE): entornos workstation
 - Java Platform, Enterprise Edition (Java EE): entornos distribuidos
 - Otras: Java Card,...
- Ver <http://java.sun.com/reference/api>



Introducción a la tecnología java

Tecnologías Java de Sun - Familia SE



Introducción a la tecnología java

Tecnologías Java de Sun - Familia SE

- Tiene dos partes:
 - El entorno de ejecución (JRE) consta de:
 - Librerías java (bytecodes de los paquetes compilados) o APIs (Application Program Interface)
 - JVM Hotspot
 - Plugins para ejecutar los applets en los navegadores
 - Entorno “Java Web Start” para descargar y lanzar aplicaciones java a través de internet sin necesidad de un navegador.
 - Documentación y Licencia
 - El entorno de desarrollo (JDK) incluye herramientas como:
Compilador javac, javadoc, depurador jdb, desensamblador javap,...
- Licencia: en Nov'06 Sun libera el compilador javac, la JVM y la herramienta de ayudas bajo licencia GPL2. Faltan algunas librerías, debido a que no fueron desarrolladas directamente por Sun. También deja bajo GPLv2 JME y JEE

Introducción a la tecnología java

Tecnologías Java de Sun - APIs

- Referencia: <http://java.sun.com/reference/api/>
- Ej. Java SE versión 6: <http://java.sun.com/javase/6/docs/api/>
- Zonas de interés en la página:
 - Arriba izda: Listado de Paquetes
 - Abajo izda: Listado de Clases e Interfaces de cada paquete
 - Derecha:
 - Visión general de todos los paquetes
 - Paquete de una clase seleccionada
 - Clase y paquetes en que se usa
 - Árbol jerárquico
 - Especificaciones anticuadas, índice y ayuda

Introducción a la tecnología java

(Otros Productos no Sun)

- Runtime:
 - JVMs:
 - Libres/open source: Kaffe, SableVM, GNU gcj/gij, Apache harmony, Blackdown (proviene de personas voluntarias de Sun),...
 - Privativas: IBM, HP, Apple, BEA Syst., ...
 - Librerías: GNU Classpath (implementación libre de las clases de Java , casi todas las de Sun versión 1.5)
- Desarrollo:
 - IDEs (Integrated Development Environment): IBM Eclipse, Sun Netbeans, jedit
 - Compiladores: GNU gcj (puede generar no sólo bytecodes sino también código nativo, ej: `gcj H1.java -main H1`), jikes
 - Otras herramientas: proyectos (ant), paquetes (fastjar), appletviewer,...

Introducción a la tecnología java

Procedimiento básico de desarrollo

- **Edición** del fuente, por ejemplo con el editor emacs:

```
emacs NombreClasePublica.java
```

- **Compilación:** un fichero fuente da origen a tantos ficheros .class como clases + interfaces contenga, sean privadas o públicas.

```
javac NombreClasePublica.java
```

*Se genera **NombreClasePublica.class***

- **Ejecución:** java NombreClasePublica

El fichero que se ejecuta es NombreClasePublica.class

- Si se usan paquetes (ver capítulo correspondiente):
 - La variable de sistema CLASSPATH debe estar al valor adecuado .
 - Para ejecutar el main de una clase no perteneciente al paquete por defecto: java nombrePaquete.NombreClasePublica



Java sintaxis básica

Sintaxis básica

Características generales

- Lenguaje “case-sensitive”: se distinguen mayúsculas de minúsculas
- Tipo de líneas de un fichero fuente: además de las de C, nuevo comentario:
 - Una o más líneas entre `/**` y `*/` para generar documentación con **javadoc** (*herramienta para generar documentación a partir de ficheros fuente*)
- Todo el código debe de estar incluido en clases
- Hay una clase principal por la que comienza la ejecución

Sintaxis básica

Variables y constantes

- Constan de:
 - Identificador o etiqueta: *nombre de la variable*
 - Valor. Valor inicial: *=xxx o por defecto*
 - Ámbito: zona del programa de validez.
 - Modificador: elemento sintáctico para indicar características como ej. ámbito de acceso, si es constante o variable,...
 - Tipo: se dividen en tipos *primitivos* y tipos *referencia*
 - No interesa la dirección en memoria.

Ejemplo:

```
public abstract class Juego {  
    private int mVidas;  
    protected int mVidasTotal;  
    static private int mRecord = 0;  
    (...)  
}
```

Sintaxis básica

Variables y constantes – Clasificación

- Atributo de un objeto o de una clase.
 - Ámbito: según modificadores de ámbito del atributo y de su clase
 - Inicialización por defecto: ver lista de tipos
- Locales a un método.
 - Ámbito: el método
 - Inicialización por defecto: no se inicializan y hay que darles algún valor antes de ser usadas; si no error de compilación
- Parámetros de un método o de un gestor de excepciones.
 - Ámbito: el método
 - Inicialización: el valor de la llamada.
- (No hay variables globales)

Sintaxis básica

Variables y constantes – Clasificación – Ejemplo:

```
public abstract class Juego {
```

```
    private int mVidas;  
    protected int mVidasTotal;  
    static private int mRecord = 0;
```

Atributos de la clase

```
    public void Juega() {  
        int num;  
        boolean salir = false;
```

Vbles locales del método

```
        mVidas=10;
```

*Se pueden utilizar desde
cualquier método de la clase*

```
        ReiniciaPartida();
```

```
    do {
```

```
        (...)
```

```
    }
```

```
}
```

```
}
```

Sintaxis básica

Variables y constantes locales – Declaración

- Sintaxis de declaración:
 - Variable local:
`<tipo> <identificador> [=<valor>];`
 - Constante local: se ha de inicializar en la declaración; después no se puede.
`final <tipo> <identificador>=<valor>;`
- (Por tanto, el único modificador válido, y que es el que establece la diferencia entre ambas, es `final`. No puede haber ej. `static`)

Sintaxis básica

Variables y constantes – Identificadores

- Normas de identificadores: idem que en C, salvo que las variables pueden empezar con o incluir \$ (no es típico)
- Convención para identificadores:
 - Las variables empiezan por una letra minúscula. Si un identificador está compuesto por más de una palabra, las palabras se ponen juntas y la inicial de la segunda, tercera,... palabra se pone en mayúscula. Ej. nombreDato
 - Las constantes se escriben en mayúsculas. Si tiene varias palabras se separan con _ Ej: NUM_MAX

Sintaxis básica

Variables y constantes – Tipos primitivos

- Hay 8 tipos primitivos:

Palabra clave	Tamaño	Comentario	Valor por defecto	Ejs. de Literales
byte	1 byte	Con signo	0	-2
short	2 bytes	Con signo	0	10000
int	4 bytes	Con signo	0	100000 032 (octal) 0x1F (hexadecimal)
long	8 bytes	Con signo	0L	3123456789
float	4 bytes	IEEE 754	0.0f	123.4f 12.4F
double	8 bytes	IEEE 754	0.0d	12.3 -124.678d 14.678D 1.234e2 1.23E4
char	2 bytes	Carácter unicode, desde '\u0000' a '\uffff' Sin signo	'\u0000'	'x' '\u0108' \b \t \n \r \f \" \' \\
boolean	1 bit o 1 byte	true / false	false	true false

- <http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>

Sintaxis básica

Variables y constantes – Tipos referencia

- Un tipo referencia indica que en la variable se guardará un objeto o instancia de una clase.
- Ocupan el n° de bytes necesarios para almacenar una dirección, y depende de la arquitectura.
- Se inicializan a null.
- Sólo puede guardar valores de un tipo, (aunque hay casos en que pueden ser de tipos distintos; ver anexo de herencia y conversiones entre tipos referencia) ej:

```
Computer pc;  
String cadena;  
cadena=pc; //ERROR
```

Sintaxis básica

Variables y constantes – Otros tipos de datos

- En Java se pueden ver variables que se pueden interpretar a primera vista como tipos de datos primitivos pero realmente se trata de referencias a objetos.
- Por ejemplo la clase String nos ofrece la posibilidad de guardar y manejar cadenas de caracteres:

```
String cadena="se guarda un texto ";  
String cadena2 = cadena.trim();
```

<http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/String.html>

Sintaxis básica

Variables y constantes – Cadenas de caracteres

- La clase String ofrece muchas posibilidades para el manejo de textos.
- Por ejemplo siempre que se indica un literal entre comillas se crea una clase String y nos ofrece todas sus posibilidades.
- Java ofrece el operador (+) para concatenar Strings:

```
String cadena1 = "Nombre" + " " + "Apellido";  
String cadena2 = " Dirección"  
String cadena3 = cadena1 + cadena2;
```

- Además el método **“toString()”** presente en todas las clases nos ofrece la posibilidad de convertir el resultado de una clase a texto. En algunos casos es llamado automáticamente para dar una salida de texto (concatenación de objetos, métodos print,...).

Sintaxis básica

Variables y constantes – Cadenas de caracteres

- Pero la clase String guarda cadenas de caracteres fijas. Una vez iniciada no se puede modificar su contenido. Se pueden tener nuevos Strings como resultados de los métodos disponibles pero el contenido no se puede modificar.
- Como alternativa existe la clase “StringBuffer” que permite modificar su contenido. Es más eficiente que String.

Ejemplo:

```
StringBuffer strB = new StringBuffer("Nombre");  
strB.append(" Apellido");
```

<http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/StringBuffer.html>

Sintaxis básica

Literales (*valores fijos indicados literalmente*)

- (los indicados en la tabla de tipos de datos): 20, 1.3 , ...
- Cadena de caracteres: entre comillas dobles, ej. "cadena"

Sintaxis básica

Operadores - Comparación con C

- Nuevos operadores binarios:
 - `op1>>>op2` Desplaza los bits de `op1` sin signo a la derecha, `op2` posiciones. Antes, si `op1` tiene menos bits que un `int` se convierte a `int`.
 - `op1=>>>op2` Idem haciendo asignación a `op1`
 - `op1 instanceof op2` Determina si `op1` es un objeto de clase `op2`
- Casting: (sólo tiene sentido entre tipos primitivos)
 - Si se va a perder precisión da error al compilar (casting obligatorio)
 - El tipo `boolean` no es convertible al resto de tipos y viceversa.
- No existen:
 - Operadores de punteros y memoria: `*` `&` `->` `sizeof`
 - Operador coma `,`

Sintaxis básica

Operadores - Tabla de precedencia

Grupo	Operador
Expresiones varias	() . [] expr++ expr--
Operadores unarios	+ - ! ~ ++expr --expr
	(typecast) new
Operadores binarios	* / %
	+ -
	>> << >>>
	< > <= >= instanceof
	== !=
	&
	^
	&&
Operador ternario	?:
Asignación	= += -= *= /= %= >>= <<= >>>= &= ^= =

Sintaxis básica

Sentencias de control de flujo - Comparación con C

- Condicionales: ejecutan una sola vez una o varias líneas de código dependiendo de una condición
 - if
 - switch
- Repetitivas: ejecutan varias veces una o varias líneas de código dependiendo de una condición
 - while
 - do – while
 - for
- En todos los casos la condición ha de ser de tipo boolean (no como en C)

Sintaxis básica

Funciones - Comparación con C

- Una función no existe por independiente sino que es parte de una clase (“método de una clase”).
- Paso de argumentos por valor (no obstante, para parámetros de tipo referencia sí puede haber modificación de contenidos apuntados)
- Sobrecarga de métodos: puede haber métodos con un mismo nombre que ejecutan distinto código (hace distintas funcionalidades), y se diferencian según el nº y los tipos de argumentos.

Ej:

En C	En Java
<code>void mostrarInt(int i)</code>	<code>void mostrar(int i)</code>
<code>void mostrarLong(long l)</code>	<code>void mostrar(long l)</code>
<code>void mostrarFloat(float f)</code>	<code>void mostrar(float f)</code>

Sintaxis básica

Otras palabras reservadas

- Java reserva estas palabras para el futuro (de momento no tienen un uso específico):

byvalue	cast	const	future
generic	goto	inner	operator
outer	rest	var	

Java funciones básicas de E/S

Funciones básicas de E/S

Salida por stdout

- El atributo out de la clase System (que está en el paquete java.lang) es de clase PrintStream (que está en el paquete java.io), la cual tiene varios métodos para sacar información por stdout, de la forma:

```
println (...)
```

Incluye salto de línea.

Ver <http://java.sun.com/javase/6/docs/api/java/io/PrintStream.html>

Ej:

```
System.out.println("Kaixo, " + " mundu");
```

Funciones básicas de E/S

Entrada por stdin

- El atributo `in` de la clase `System` (que está en el paquete `java.lang`) es de clase `InputStream` (que está en el paquete `java.io`), la cual tiene varios métodos para leer por `stdin`, de la forma:

`read (...)` throws `IOException`

Ver <http://java.sun.com/javase/6/docs/api/java/io/InputStream.html>

Ej:

```
public static void main (String [] args) throws IOException {  
    byte b[]=new byte[12];  
    System.in.read(b, 0, b.length);  
}
```

Otra clase de interés en la versión 6: `java.lang.Console`



Java programa mínimo

Programa mínimo

- La ejecución de un programa en Java comienza por el método **main()** que puede estar incluido en cualquier clase que forme parte del programa.
- El prototipo de la función main debe ser:
`public static void main(String args[])`
- En args[] se pueden recibir parámetros en la línea de comando en modo texto. Si se requieren datos numéricos hay que convertirlos.
- Por ejemplo si hacemos un programa que se llame miCiudad y le pasamos como parámetro “Bilbao” la línea de comando sería:

`java miCiudad Bilbao`

y el código

```
public class miCiudad {  
    public static void main( String args[] ) {  
        System.out.println( "Mi ciudad es " + args[0]);  
    }  
}
```

Programa mínimo

- La ejecución de un programa en Java comienza por el método **main()** que puede estar incluido en cualquier clase que forme parte del programa.
- El prototipo de la función main debe ser:
`public static void main(String[] args)`
- Un programa básico que muestre un texto por pantalla podría ser:

```
public class HolaJava {  
    public static void main( String[] args ) {  
        System.out.println( "Hola Java!" );  
    }  
}
```




Java clases y objetos

Clases y Objetos

Introducción

- Clase: abstracción de las entidades de la vida real, incluyendo:
 - Atributos o características de la clase
 - Métodos u operaciones realizables sobre la clase. En java no existe código fuera de los métodos.(Miembros = atributos + métodos)
- Nomenclatura por convenio:
 - Las clases comienzan por letra mayúscula.
 - Los métodos empiezan por minúscula.
 - Si un identificador tiene más de una palabra, se ponen juntas y la inicial de la segunda, tercera,... palabra va en mayúscula. Ejs. ClasePrincipal, abrirPuerta,...
 - Los atributos siguen el criterio dado para variables y constantes

Clases y Objetos

Definición de una clase

- Sintaxis:

```
<declaración de la clase> {  
    <cuerpo de la clase>  
}
```

(la primera llave puede ir en la siguiente línea)

siendo <declaración de la clase>:

```
[<modificadoresClase>] class <NombreClase>  
[extends <NombreSuperclase>] [implements <ListaInterfaces>]
```

(extends e implements se verán más adelante)

y <cuerpo de la clase>:

- Declaración de atributos
- Declaración y cuerpo de métodos

Clases y Objetos

Declaración de una clase – modificadores de clase

- Modificadores:
 - Si hay varios se separan por espacios
 - Posibles valores:
 - Modificador de ámbito: public
 - Otros modificadores: uno de éstos (son excluyentes): abstract, final
- public:
 - Sólo las clases declaradas así pueden ser accedidas desde clases e interfaces que estén en otros paquetes. Y si no es public la clase puede ser accedida sólo desde las clases e interfaces del paquete al que pertenece, pero no desde las de otro paquete
 - Toda clase pública ha de estar definida en un fichero fuente con nombre coincidente con el de la clase. Por tanto, no puede haber otras clases públicas o interfaces públicas en un fichero fuente.

Clases y Objetos

Declaración de una clase – modificadores de clase

- **abstract:**
 - Define clases que no pueden ser instanciadas. Su uso es como superclases de otras.
 - Si una clase es abstract puede tener métodos no abstract, pero si algún método es abstract la clase ha de ser abstract
- **final:**
 - De ella no se pueden derivar subclases. Utilidad ej. seguridad.
 - Si algún método de una clase es final, la clase no tiene por qué ser final

Clases y Objetos

Cuerpo de una clase – declaración de atributos

- Sintaxis para cada atributo:

`[<modificadores de atributo>] <tipo> <nombreAtributo>[=valor];`

- Modificadores:

- Si hay varios se separan por espacios
- Posibles valores:
 - Modificadores de ámbito: un valor de entre éstos:
public, protected, friendly (o ninguno), private
 - Otros modificadores: uno o varios de entre éstos:
static, final, transient, volatile

Clases y Objetos

Cuerpo de una clase – modificadores de atributos

- **public:**
 - El atributo es accesible desde (métodos de) cualquier clase, siempre que no tenga una restricción por modificador de clase no public.
- **protected:**
 - El atributo puede ser accedido desde (métodos de) subclases que lo heredan y también desde (métodos de) clases del mismo paquete
- **friendly:**
 - El atributo puede ser accedido desde (métodos de) clases del mismo paquete
 - Es el valor por defecto de modificador de ámbito
- **private:**
 - El atributo sólo puede ser accedido desde (métodos de) la misma clase

Clases y Objetos

Cuerpo de una clase – modificadores de atributos

- final:
 - El valor del atributo es constante y se inicializa en su declaración.
- static:
 - Es un atributo de clase (no de instancia), y almacena el mismo valor para todos los objetos de la clase.
 - No es necesario crear un objeto de la clase para acceder al atributo, y también se puede acceder al mismo invocando: `<NombreClase>.<nombreAtributo>`
- transient: (poco usado en esta asignatura)
 - Indica que el atributo almacena información no persistente del objeto, y en un proceso de almacenamiento no se guardaría.
- volatile: (poco usado en esta asignatura)
 - El atributo almacena valores modificables por distintos threads.

Clases y Objetos

Cuerpo de una clase – métodos

- En una clase puede haber varios métodos con igual nombre que se diferencien en la lista de parámetros (nº o tipos).
- Sintaxis:

```
<declaración del método> [{  
    <cuerpo del método>  
}] [;]  
(la primera llave puede ir en la siguiente línea)
```

siendo <declaración del método>:

```
[<modificadoresMétodo>] [<tipo>] <nombreMétodo>  
([<listaParámetros>])[throws <ListaExcepciones>]
```

(throws se verá más adelante)

Clases y Objetos

Cuerpo de una clase – modificadores de métodos

- Modificadores:
 - Si hay varios se separan por espacios
 - Posibles valores:
 - Modificadores de ámbito:
 - Un valor de entre éstos (o ninguno equivalente a friendly): public, protected, friendly, private
 - Se comportan de forma idéntica al caso de atributos
 - Otros modificadores: uno o varios de entre éstos:
 - static, abstract, final, native, synchronized
 - No se permiten estas combinaciones:
 - abstract y final
 - abstract y static

Clases y Objetos

Cuerpo de una clase – modificadores de métodos

- **abstract:**
 - Sólo se debe especificar su declaración o prototipo, pero no el cuerpo que se reemplaza por ;
 - Si hay al menos un método abstract la clase ha de ser abstract
 - El sentido de un método abstract es declarar un método en cuanto a parámetros, pero dejar que las subclases lo implementen.
 - (Más en capítulo de Herencia)
- **final:**
 - El método no puede ser redefinido en subclases.

Clases y Objetos

Cuerpo de una clase – modificadores de métodos

- **static:**
 - Es un método de clase (no de instancia)
 - No es necesario crear un objeto de la clase para acceder al método, y también se puede acceder al mismo invocando:
`<NombreClase>.<nombreMétodo>`
 - Si el método se redefine en subclases en éstas ha de ser static.
 - Sólo puede acceder a atributos y métodos static.
- **native:** (poco usado en esta asignatura)
 - Escrito en un lenguaje de programación distinto a java (ej. C).
 - El método no tiene cuerpo.
- **synchronized:** (poco usado en esta asignatura)
 - Sólo puede haber un método synchronized en ejecución para cada clase. Tiene una función de bloqueo. Ej. de uso: en threads.

Clases y Objetos

Cuerpo de una clase – tipo y parámetros de métodos

- tipo: una de estas posibilidades:
 - El tipo devuelto (con return) o bien void si no devuelve nada.
 - Si se indica un tipo y no se devuelve un valor del mismo da error en compilación.
 - En los métodos constructores no hay que poner nada (ni void)
- Lista de parámetros: idem que en C. No hace falta poner void si no recibe ninguno.

Clases y Objetos

Cuerpo de una clase – cuerpo de métodos

- Instrucciones entre { y } salvo si el método es abstract en cuyo caso se sustituyen por ;
- El cuerpo de un método puede estar vacío (ninguna instrucción entre { y }) pero no por ello es abstract.
- Tendrá o no `return` según devuelva algo o no.

Clases y Objetos

Acceso a atributos y a métodos

- Si se accede a atributos/métodos de la misma clase, sean static o no:

`<nombreAtributo>`

`<nombreMetodo>([params])`

- Si se accede a atributos/métodos de otra clase:

`<objeto>.<nombreAtributo>`

`<objeto>.<nombreMetodo>([params])`

- Si se accede a atributos/métodos static de otra clase, otra posibilidad es:

`<NombreClase>.<nombreAtributo>`

`<NombreClase>.<nombreMetodo>([params])`

- Otras posibilidades (más adelante): con `this` y `super`

Clases y Objetos

Método main

- Todo programa ejecutable standalone (no applet) ha de tener una clase pública en la que se declare un método llamado main de esta forma:

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

- Comentarios:
 - public: por acceder desde fuera del paquete
 - static: para acceder llamándole desde la clase (no desde un objeto concreto). Por tanto, no podría llamar a métodos no estáticos de la misma clase.
 - Con los argumentos indicados (si no, sería un método diferente): array de strings

Clases y Objetos

Constructores

- Son métodos especiales de una clase que sirven para inicializar objetos.
- Siempre son llamados durante la creación de los mismos, explícitamente (con código) o implícitamente (de forma interna desde la JVM).
- El constructor no crea nuevas instancias de clases, sino que sólo las inicializa.
- Puede haber más de un constructor, diferenciándose entre ellos en la lista de parámetros, gracias a la propiedad de polimorfismo.

Clases y Objetos

Constructores – Sintaxis

- Sintaxis del constructor:

```
[<modificadoresMétodo>] <NombreClase>  
    ([<listaParámetros>]) [throws<ListaExcepciones>] {  
<cuerpo del constructor>  
}
```

- Diferencia con otros métodos:
 - No se indica tipo
 - El nombre coincide con el de la Clase

Clases y Objetos

Constructores – Sintaxis

- Cuerpo del constructor:
 - Un constructor no devuelve nada (pero podría emplear `return` sin ninguna expresión a continuación para, ej. salir del método).
 - Desde un constructor es posible invocar a otro constructor de la misma clase (ver usos de `this`) o de la clase inmediatamente superior (ver herencia). Condiciones:
 - Sólo puede hacerse la llamada en la primera línea. Por tanto no puede haber más de una llamada a un constructor de los indicados.
 - La lista de argumentos no puede incluir referencias a atributos de la clase.

Clases y Objetos

Constructores – Constructor por defecto

- Es aquél al que no se le pasa ningún parámetro.
- Si no se define ningún constructor para una clase, Java crea el constructor por defecto (por eso no es obligatorio definir el constructor de cada clase), el cual inicializa todos los atributos a sus valores por defecto. Ej:

```
public class Point {int x, y;}
```

equivale a:

```
public class Point {  
    int x, y;  
    public Point() { super(); }  
}
```

- Excepción a lo anterior: si se define algún otro constructor para una clase, Java no crea el constructor por defecto y es necesario definirlo si se quiere usar.

Clases y Objetos

Constructores – invocación

- Para invocar a un constructor (el que es por defecto u otros):

`Clase()`

`Clase(<lista1Parámetros>)`

`Clase(<lista2Parámetros>)`

`...`

- Puede que algunos constructores de una clase sean accesibles y otros no. Habrá que inicializar objetos invocando al constructor adecuado.
- No se puede invocar a un constructor con `this` o `super` si no es desde otro constructor de la misma clase

Clases y Objetos

Creación de objetos

- Para crear un objeto de una clase se siguen dos pasos:
 - Declaración de su tipo referencia. Con esto se crea una referencia: se reserva memoria para almacenar sólo la referencia.
 - Creación e inicialización llamando a algún constructor mediante `new Clase(<listaParámetros>)`:
 - Mediante `new` la JVM reserva memoria para almacenar el nuevo objeto.
 - Mediante el constructor el nuevo objeto es inicializado.

Ej:

```
ClaseA objeto1, objeto2;  
objeto1 = new ClaseA();  
objeto2 = new ClaseA(parm1, parm2);  
...
```

Clases y Objetos

Asignación y copia de objetos

- Al emplear el operador asignación entre referencias a objetos:
 - Se asigna una nueva referencia al objeto representado por la referencia origen (quedando dos referencias apuntando al mismo).
 - El objeto antes apuntado por la referencia destino puede que ya no sea útil (ej. si no tiene referencias que apunten al mismo), en cuyo caso el garbage collector liberará su memoria.
 - No se crea un nuevo objeto (para esto, habría que emplear un método llamado `clone`).

Ej:

```
Punto a1,a2; //se declaran dos referencias a objetos
a1=new Punto(4,6); //se crea el primer objeto
a2=new Punto(3,42); //se crea el segundo objeto
a1=a2; //el antiguo objeto Punto(4,6) es desechable
```

Clases y Objetos

Asignación y Conversión entre Tipos referencia distintos

- Es posible hacer asignaciones y conversiones sin casting entre objetos de tipos referencia distintos cuando el origen (derecha) descienda de/implemente el destino (izquierda).
- En todos los casos sólo se podrá acceder a los métodos de la superclase o superinterface (a la que pertenece t), y no a otros.

Clases y Objetos

Asignación y Conversión entre Tipos referencia distintos

- Casos: (s: origen t: destino)
 - Si s es de una clase derivada de la de t. Ej:
`Transporte t; //podría haber sido ej. Object`
`Bici s;`
`t=s; //No es necesario t=(SuperClase)s;`
 - Si t es de tipo interface y s es de una clase o de un interface que implementa el interface de t. Ej:
`I t;`
`ClaseQueImplementaI s1;`
`t=s1;`
`InterfaceQueImplementaI s2;`
`t=s2;`
 - Si t es de clase Object y s es cualquier array, clase o interface.

Clases y Objetos

Usos de this

- `this` es una referencia al propio objeto instanciado
- No puede aparecer dentro de un método `static`
- Usos:
 - Para acceder a un atributo de la misma clase, en el caso de que el nombre coincida con algún parámetro o variable local del método desde el que se accede. Ej:

```
public int metodo(int x) {  
    this.x=2*x; //siendo x un miembro  
}
```

- Como parámetro de un método. Ej: `c.sumar(this);`
- Para invocar a un constructor desde otro constructor de la misma clase, ej:

```
public Clase(int x, int y) {this(x,y,true); }
```

Clases y Objetos

Clases y ficheros fuente

- Un fichero fuente:
 - Puede contener una clase pública. En este caso el nombre del fichero ha de ser de la forma NombreClasePublica.java
 - Puede no contener ninguna clase pública. En este caso el nombre del fichero no importa.
 - En cualquiera de los casos anteriores el fichero puede contener clases no públicas, que serían accesibles sólo para las clases del mismo paquete.
 - Nunca puede contener más de una clase pública.

Clases y Objetos

(Anexo: Varios clases)

- Inicializadores de clase y de instancia
(JLS 8.6 Instance initializers, 8.7 Static initializers)
- Tipos de clases:
 - Clase top-level: las que están fuera de otras clases y de métodos
 - Clase local: dentro de un método
 - Clase miembro o anidada: si una clase es un atributo de otra clase.
 - Clase anónima

Clases y Objetos

(Anexo: Clases anidadas)

- Usos:
 - Agrupación lógica
 - Mayor encapsulación, ej. acceso a los miembros de una clase sólo desde otra
- Clasificación:
 - Estáticas. Creación de un objeto:
`ClaseExterna.ClaseEstatica o = new ClaseExterna.ClaseEstatica();`
 - No estáticas o inner. Creación de un objeto:
`ClaseExterna.ClaseInner o = objetoExterno.new ClaseInner();`
- Clase inner local: declarada en el cuerpo de un método.
- Clase inner anónima: idem sin nombrarla

Clases y objetos - ejemplos

- Valores por defecto:
 - Crear una clase con tres atributos int, float y char.
 - Crear un objeto de esa clase y mostrar sus valores.
- Referencia copiada:
 - Crear una clase con tres atributos int, float y char.
 - Crear un objeto de esa clase.
 - Crear una segunda referencia a esa clase y copiar el valor de la referencia anterior utilizando el operador de asignación.
 - Modificar los valores de los atributos desde la referencia 2.
 - Mostrar los valores de la referencia 1.
- Constructor copia:
 - Crear una clase con tres atributos int, float y char.
 - Añadir un constructor copia para clonar los objetos y repetir el ejemplo anterior.



Java herencia

Herencia

Introducción

- Definición:
 - posibilidad de que una clase adquiriera atributos y/o métodos de otra ya existente,
 - pudiendo además añadirse otros atributos y/o métodos,
 - y redefinirse métodos heredados.
- A la clase madre se le llama “superclase” y a la hija “subclase”.
- Utilidad: reutilización de código en base a que unas clases se parecen a otras en sus atributos y métodos.
- Las clases superiores describen un comportamiento más general y las subclases más específico.

Herencia

Introducción – herencia en Java

- Herencia simple: una clase tiene una y sólo una superclase (que será Object si no se explicita ninguna).
- Se heredan todos los atributos y métodos a excepción de los constructores.
- El acceso a atributos y métodos de la superclase se puede limitar mediante modificadores de ámbito.
- Árbol API Sun JSE:
<http://java.sun.com/javase/6/docs/api/java/lang/package-tree.html>

Herencia

Sintaxis y limitaciones

- Para indicar que una clase hereda de otra se emplea esta sintaxis:

```
...class <Subclase> extends <SuperClase>...
```
- Condición: en la superclase ha de haber algún constructor accesible para la subclase. Casos:
 - Si ambas clases están en el mismo paquete, no importa si la superclase es o no public, pero la superclase ha de tener algún constructor accesible, es decir no private. Puede ser el constructor por defecto u otro.
 - Si ambas clases no están en el mismo paquete, la superclase debe ser public y además ha de tener algún constructor accesible, es decir public o protected.
- Una clase final no puede tener subclases

Herencia

Clase Object

- Todas las clases en java descienden de la clase `java.lang.Object` (aunque el programador no lo indique con `extends`), siendo dicha clase la raíz de todas las demás.

Definición:

<http://java.sun.com/javase/6/docs/api/java/lang/Object.html>

- Métodos de interés redefinibles por el programador:

```
public boolean equals(Object obj)
```

```
public String toString()
```

- Métodos de interés no redefinibles por el programador:

```
public final Class<?> getClass()
```

Herencia

Redefinición de métodos

- Consiste en crear un método en la subclase coincidente en nombre y parámetros con otro método de la superclase
- Condiciones:
 - No se pueden redefinir los métodos final
 - Si un método static se redefine en subclases en éstas ha de ser también static.
 - Si un método no static se redefine en subclases en éstas no puede ser static.
 - Los métodos redefinidos pueden ampliar los derechos de acceso pero nunca restringirlos.

Herencia

Acceso a métodos de la clase superior

- Si se redefine un método, es posible llamar al método original de la superclase inmediatamente superior (no más niveles) mediante:

`super.<nombreMetodo>([parms])`

- Han de cumplirse estas condiciones:
 - Que el método de la superclase sea accesible
 - Que no se use `super` desde un método `static`

Herencia

Clases y métodos abstract

- Una clase que derive de otra clase abstract con métodos abstract podrá:
 - Redefinir todos ellos.
 - No redefinir todos ellos. En este caso, la subclase también habrá de ser abstract, y respecto a los métodos abstract que se hayan heredado pero no redefinido hay dos posibilidades:
 - Volver a declararlos con abstract igual que en la superclase.
 - No declararlos.

Constructores en clases derivadas

- Una clase no hereda los constructores de su superclase.
- Una clase puede llamar a los constructores de su superclase de la forma: `super([params])`

En caso de que se haga esta llamada:

- Sólo se puede hacer desde un constructor (no desde cualquier método).
- El constructor invocado ha de ser accesible.
- Debe ser la primera sentencia del cuerpo de dicho método, y en el resto de líneas del método no puede llamarse a otro constructor de la misma clase o de la superclase.
- La lista de argumentos no puede incluir referencias a atributos de la clase.

Constructores en clases derivadas

- Si en el constructor de una clase el programador no incluye una llamada a algún constructor de la superclase, Java incluye de forma automática una llamada al constructor por defecto de la superclase, y si éste no existe hay un error de compilación.
- Así, cada vez que se inicializa un objeto cualquiera:
 - Se va llamando a los constructores de las superclases, bien de forma explícita o bien al constructor por defecto de la superclase.
 - Se llega hasta la clase Object, donde se inicializa un objeto de la clase Object.
 - Después, se ejecutan hacia abajo los constructores del resto de la jerarquía de clases, que van añadiendo las características de cada una.

Herencia

(Anexo: Modificador protected)

- Si un atributo o método de una superclase A tiene el modificador protected, entonces el acceso al mismo desde una subclase B de otro paquete será o no posible según cómo se haga la referencia para intentar acceder:
 - Sí es posible:
 - `<atributo>` y `this.<atributo>`
 - `super.metodo([<params>])` y `super([<params>])`
 - No es posible:
 - `objeto.<atributo>` siendo objeto de la clase A
 - `objeto.metodo([<params>])` siendo objeto de la clase A. Tampoco es posible si objeto es de una clase C que sea subclase de A y hermana de B, y el método no está redefinido en C como accesible para B.
 - `new A([<params>])` siendo el constructor A protected

Herencia

(Anexo: Redefinición de atributos)

- Si al hacer una subclase o subinterface se añaden nuevos atributos de igual nombre a los ya existentes, (redefinición de atributos), se distinguen ambos valores, de modo que:

`super.<atributo> == ((SuperClase)this).<atributo> != <atributo>`

- El acceso a atributos no depende de la clase en tiempo de ejecución; el acceso a métodos sí.

Ej:

```
class S { int x = 0; int z(){ return x;} }
class T extends S { int x = 1; int z(){ return x;} }
class Test {
    public static void main(String[] args) {
        T t = new T(); //t.z() y t.x valen 1
        S s = new S(); //s.z() y s.x valen 0
        s = t; //s.z() vale 1 pero s.x vale 0 !!!
    }
}
```



Declaración de una clase – modificadores de clase

- Modificadores:
 - Si hay varios se separan por espacios
 - Posibles valores:
 - Modificador de ámbito: public
 - Otros modificadores: uno de éstos (son excluyentes): abstract, final
- public:
 - Sólo las clases declaradas así pueden ser accedidas desde clases e interfaces que estén en otros paquetes. Y si no es public la clase puede ser accedida sólo desde las clases e interfaces del paquete al que pertenece, pero no desde las de otro paquete
 - Toda clase pública ha de estar definida en un fichero fuente con nombre coincidente con el de la clase. Por tanto, no puede haber otras clases públicas o interfaces públicas en un fichero fuente.



Declaración de una clase – modificadores de clase

- **abstract:**
 - Define clases que no pueden ser instanciadas. Su uso es como superclases de otras.
 - Si una clase es abstract puede tener métodos no abstract, pero si algún método es abstract la clase ha de ser abstract
- **final:**
 - De ella no se pueden derivar subclases. Utilidad ej. seguridad.
 - Si algún método de una clase es final, la clase no tiene por qué ser final



Recordatorio: Modificadores de métodos

- **abstract:**
 - Sólo se debe especificar su declaración o prototipo, pero no el cuerpo que se reemplaza por ;
 - Si hay al menos un método abstract la clase ha de ser abstract
 - El sentido de un método abstract es declarar un método en cuanto a parámetros, pero dejar que las subclases lo implementen.
 - (Más en capítulo de Herencia)
- **final:**
 - El método no puede ser redefinido en subclases.
- **static:**
 - Es un método de clase (no de instancia)
 - No es necesario crear un objeto de la clase para acceder al método, y también se puede acceder al mismo invocando: `<NombreClase>.<nombreMétodo>`
 - Si el método se redefine en subclases en éstas ha de ser static.
 - Sólo puede acceder a atributos y métodos static.

Herencia - Ejemplos

- Crear una clase base y una clase derivada de esta y observar como se pueden modificar los atributos y utilizar los métodos de la base desde la derivada.
- Llamada al constructor de la base desde una instancia de la derivada.
- Modificar el acceso de los atributos de la base y observar qué ocurre desde la derivada.
- Redefinir un método de la clase base en una clase derivada. Llamar desde una instancia de la derivada al método de la base y de la derivada.



Java interfaces

Interfaces

Introducción

- El sentido de uso de interfaces es corregir la imposibilidad de herencia de varias clases que hay en java.
- Una interface es un conjunto de constantes y métodos abstractos que se implementarán en clases.
- En una interface se establecen las funcionalidades (el “qué”), y en las clases se implementan (el “cómo”) las mismas, de modo que puede haber distintas implementaciones de una misma funcionalidad (métodos de igual nombre y lista de parámetros en clases distintas).
- La relación entre ficheros fuente e interfaces (nombres del fuente,...) es idéntica a la de las clases.

Interfaces

Definición de una interface

- Sintaxis:

```
<declaración de la interface> {  
    <declaración de atributos>  
    <declaración de métodos y ;>  
}
```

(la primera llave puede ir en la siguiente línea)

siendo <declaración de la interface>:
[public] interface <NombreInterface>
[extends <ListaSuperinterfaces>]

Interfaces

Declaración de una interface

- **public:**
 - Sólo las interfaces declaradas así pueden ser accedidas desde clases e interfaces que estén en otros paquetes. Y si no es public la interface puede ser accedida sólo desde las clases e interfaces del paquete al que pertenece, pero no desde las de otro paquete
 - Toda interface pública ha de estar definida en un fichero fuente con nombre coincidente con el del interface. Por tanto, no puede haber más clases públicas o interfaces públicas en un fichero fuente.
- **extends <ListaSuperInterfaces>:**
 - lista separada por , de las superinterfaces de las que hereda.
 - No es posible extender varias interfaces que tengan métodos iguales en nombre, nº y tipo de parámetros, pero devuelvan distintos tipos de resultado

Interfaces

Declaración de atributos y métodos

- Atributos:
 - Sintaxis para cada atributo:
`[public static final] <tipo> <NOMBRE_ATRIBUTO>=valor;`
 - Todos los atributos son public static final aunque no se indique. No pueden tener otro modificador. Es obligatorio inicializarlos.
- Métodos:
 - Sintaxis para cada método:
`[public abstract] <tipo> <nombreMétodo>([<listaParámetros>])
[throws <ListaExcepciones>;`
 - Todos los métodos son public abstract aunque no se indique. No pueden tener otro modificador.

Interfaces

Implementación de Interfaces en Clases

- Para indicar que una clase implementa una o más interfaces, se indica:

```
...class <Clase> implements <ListaInterfaces>...
```

(las interfaces de la lista van separadas por ,)

- No es posible si varias interfaces tienen métodos iguales en nombre, n° y tipo de parámetros, pero devuelven distinto tipo de resultado.

Interfaces

Implementación de Interfaces en Clases

- Si una clase se declara con implements:
 - Hereda todos los atributos de las interfaces.
 - Si varias interfaces tienen atributos con igual nombre, la clase no podrá hacer referencia a dichos atributos.
 - Respecto a los métodos declarados en cada una de las interfaces, hay dos posibilidades:
 - Redefinir todos ellos.
 - No redefinir todos ellos. En este caso, la subclase también habrá de ser abstract, y respecto a los métodos heredados no redefinidos hay dos posibilidades:
 - Volver a declararlos con abstract igual que en la superclase.
 - No declararlos.
 - Los métodos heredados y declarados han de ser public.

Interfaces

Extensión de Interfaces en (Sub)interfaces

- Para indicar que una interface extiende una o más interfaces, se indica:

```
...interface <I> extends <ListaInterfaces>...
```

(las interfaces de la lista van separadas por ,)

- No es posible si varias interfaces tienen métodos iguales en nombre, n° y tipo de parámetros, pero devuelven distinto tipo de resultado.

Interfaces

Objetos de tipo interface

- Se pueden declarar variables cuyo tipo sea una interface
- [Ver **Asignación y Conversión entre Tipos referencia distintos**]

Interfaces

Interface versus Clase abstract pura

- Una interface es semejante a una clase abstract con todos sus métodos abstract, pero son distintas (si no, no se necesitarían las interfaces). Diferencias:
 - Los modificadores de los miembros de una clase abstract pura pueden ser, además de las posibilidades de los interfaces:
 - Atributos: private/friendly/protected y no static/no final.
 - Métodos: private/friendly/protected y static.
 - Herencia múltiple: una clase sólo puede derivar de una única clase abstract pura, pero puede implementar varios interfaces.
- Una diferencia entre interfaces y clases abstract en general es que éstas pueden tener algún método no abstract (en el caso de que no sean abstract “puras”)



Java polimorfismo

Polimorfismo

- Polimorfismo: múltiples formas
- Permite realizar una programación genérica utilizando nombres genéricos: `Mostrar()`, `Jugar()`, `Enviar()`, ... pero al mismo tiempo al aplicarlos a distintas clases dan resultados diferentes.
- “`Mostrar()`” puede mostrar un triángulo o un rectángulo según a qué clase se aplique.

Polimorfismo

- El polimorfismo se manifiesta en Java de tres formas:
 - Métodos de una misma clase con el mismo nombre y parámetros diferentes.
 - Métodos con el mismo nombre en clases diferentes.
 - Variables tipo referencia que pueden referenciar a objetos de distinto tipo (siempre compatibles).

```
public class classA {  
    public classA(int a)  
    {  
        ...  
    }  
    public classA(int a, int b)  
    {  
        ...  
    }  
}
```

```
public class classA {  
    public static void main (String[] args)  
    {  
        ClassBase myBase;  
        myBase=new ClassDerivada();  
    }  
}
```



Java Arrays y Strings

Arrays y Strings

Arrays

- Son estructuras que almacenan en una variable múltiples valores del mismo o distinto tipo.
- Se clasifican en:
 - Estáticos: el tamaño es constante en tiempo de ejecución
 - Dinámicos: el tamaño puede variar en tiempo de ejecución
- Sus elementos empiezan en el índice 0 (como en C)
- Si se intenta acceder fuera del rango adecuado de índices se lanza la excepción `ArrayIndexOutOfBoundsException`

Arrays y Strings

Arrays estáticos

- Su tamaño no puede cambiar en tiempo de ejecución.
- Son estructuras que almacenan en una variable múltiples valores de igual o distinto tipo:
 - Si el tipo de los valores es primitivo, han de ser de un solo tipo.
 - Si el tipo de los valores es referencia, pueden ser de ser de tipos distintos (siempre compatibles).
- Internamente son de una clase para cada tipo de elementos, que cuenta con:
 - Atributo `public final int length` para indicar el nº elementos.
 - (Método `public Object clone()` para copiar un array a otro)

Arrays y Strings

Arrays estáticos de tipos primitivos

- Sintaxis en varios pasos:
 - 1. Declaración del array: se reserva espacio para la referencia del array. Dos posibilidades:

```
<tipoPrim> <nombreArray> [];
```

```
<tipoPrim> [] <nombreArray>;
```
 - 2. Creación del array: se reserva espacio para sus elementos. Dos posibilidades:

```
<nombreArray>=new <tipoPrim> [<tamaño>];
```

```
En la declaración añadir = {<elem1>,<elem2>,...<elemN>};
```
- Los pasos se pueden combinar, ej.

```
<tipoPrim> <nombreArray>[]=new <tipoPrim> [<tamaño>];
```

Nota: [] no significa opcional sino los caracteres [y] como tales

Arrays y Strings

Arrays estáticos de tipos referencia

- Sintaxis en varios pasos:
 - 1. Declaración del array: se reserva espacio para la referencia del array. Dos posibilidades:

```
<Clase> <nombreArray> [];
```

```
<Clase> [] <nombreArray>;
```
 - 2. Creación del array: se reserva espacio para sus elementos, que son referencias. Posibilidades:

```
<nombreArray>=new <Clase> [<tamaño>];
```

```
En la declaración añadir = {<obj1>, <obj2>, ...};
```
 - Los objetos pueden existir o crearse en la creación del array. ej:

```
<nombreArray>[<elem>]=new <Clase>([params]);
```

```
En la declaración añadir = {new <Clase>([params]), ...};
```
- Los pasos se pueden combinar, ej.

```
<Clase> <nombreArray>[] = new <Clase> [<tamaño>];
```


Arrays y Strings

Arrays estáticos de tipos referencia distintos

- Es posible tener arrays estáticos con objetos de tipos referencia distintos.
- Según [**Asignación y Conversión entre Tipos referencia distintos**], se consigue declarando el tipo del array de una de estas formas:
 - Que sea una interface implementada por todas las clases o interfaces de los objetos a contener.
 - Que sea una superclase de la que todos los tipos de los objetos hereden. Esto incluye el caso de declarar un array estático de tipo Object.

Arrays y Strings

Arrays estáticos multidimensionales

- Son arrays de más de una dimensión.
- Declaración del array (para 2 dimensiones). Posibilidades:

```
<tipo> <nombreArray> [][];
```

```
<tipo> [][] <nombreArray>;
```

- Creación del array (para 2 dimensiones):

```
<nombreArray>=new <tipo> [<tamaño1>][<tamaño2>];
```

- Declaración y creación en un solo paso:

```
<tipo> <nombreArray>[][]={{a1, a2, ...}{b1, b2, ...}...};
```

Siendo {a1, a2, ...} los elementos de la 1ª fila,...

- En caso de que tipo no sea primitivo sino referencia, es necesario crear los elementos a1,a2,...,b1,b2,...

Arrays y Strings

Arrays dinámicos – clase `java.util.Vector`

- Su tamaño puede cambiar en tiempo de ejecución, añadiendo o quitando elementos.
- Se emplea la clase `java.util.Vector` (es necesaria su importación):
 - Sólo puede contener objetos de tipo referencia (no primitivo).
 - El índice de sus elementos comienza en 0
 - Puede incluir objetos de distintas clases.
 - Distingue entre tamaño y capacidad:
 - Tamaño: el nº de elementos que contiene en un momento dado. Cambia al insertar o eliminar elementos.
 - Capacidad: previsión del nº máximo de elementos que contendrá. Cuando el tamaño sobrepasa la capacidad, ésta se aumenta automáticamente en la cantidad indicada por el atributo `capacityIncrement`. Si éste es ≤ 0 la capacidad se aumenta al doble.

Arrays y Strings

Arrays dinámicos – clase `java.util.Vector`

- Atributos de interés:
 - `capacityIncrement`
- Métodos de interés:
(algunos pueden lanzar `ArrayIndexOutOfBoundsException`)
 - Constructores: `Vector()`, `Vector(int initialCapacity)`, `Vector(int initialCapacity, int capacityIncrement)`
 - Añadir: `add`, `addElement`, `insertElementAt`
 - Reemplazar: `set`, `setElementAt`
 - Eliminar: `clear`, `remove`, `removeAllElements`, `removeElement`, `removeElementAt`
 - Buscar: `contains`, `elementAt`, `indexOf`, `lastIndexOf`
 - Otros: `capacity`, `isEmpty`, `size`, `setSize`

Arrays y Strings

Más clases para controlar arrays dinámicos:

- Además de Vector hay toda una familia de clases e interfaces que pueden ayudar a la utilización de arrays dinámicos.
- Esta familia de clases se puede agrupar por el interface que ofrecen. Se distinguen dos grupos principales de interfaces:
 - Collection
 - Map

que a su vez ofrecen otros subinterfaces.

- **Collection**: grupos de objetos organizados “en fila” ordenados o no.
- **Map**: grupos de objetos con una palabra clave “key” asociados a una información. La palabra clave no se puede repetir de un objeto a otro y se puede utilizar para localizar un elemento del conjunto.

Arrays y Strings

Más clases para controlar arrays dinámicos:



Collection:
Objetos “en fila”

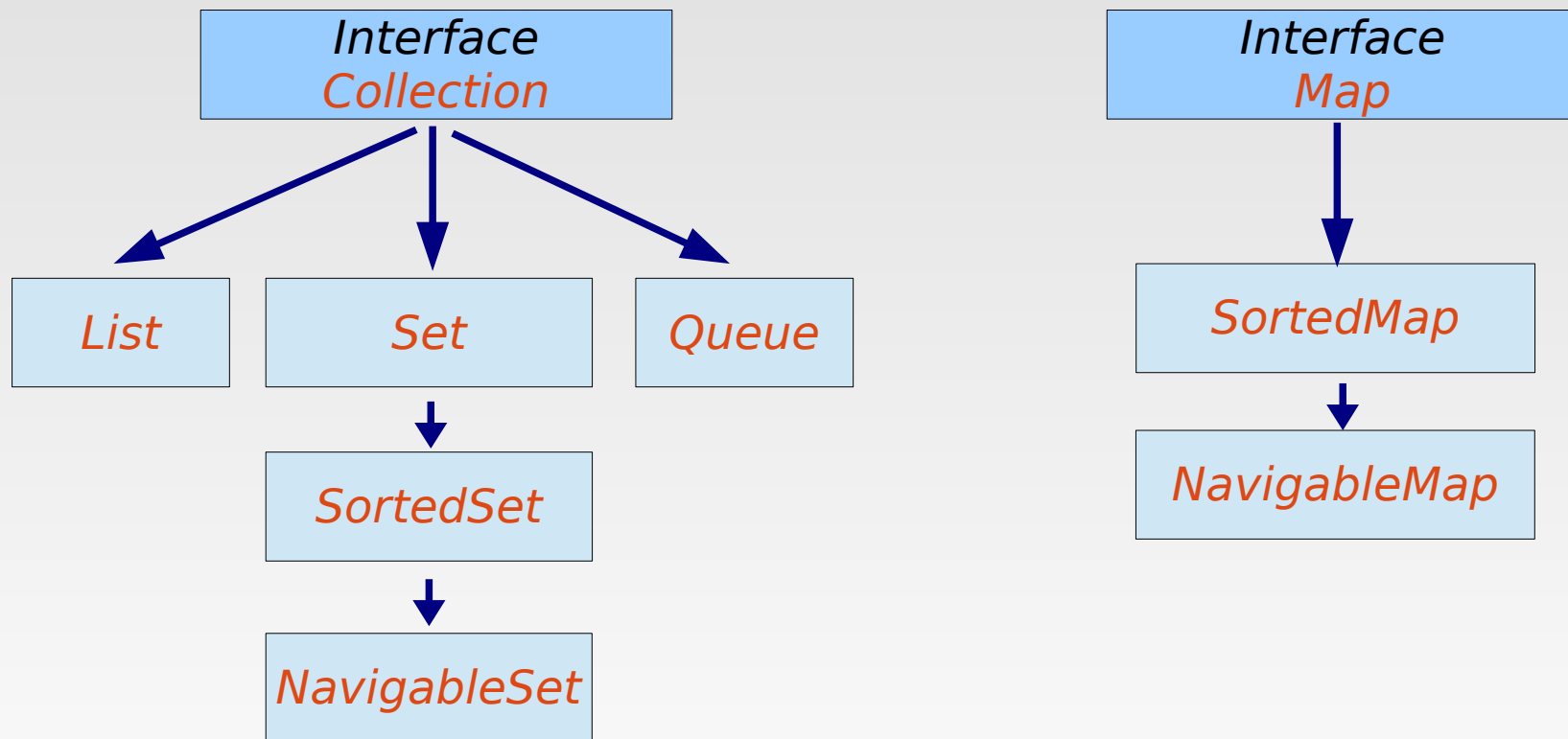


Map:
Objetos organizados en
torno a un “key”

Arrays y Strings

Más clases para controlar arrays dinámicos:

Las **interfaces** más importantes se organizan de la siguiente manera:

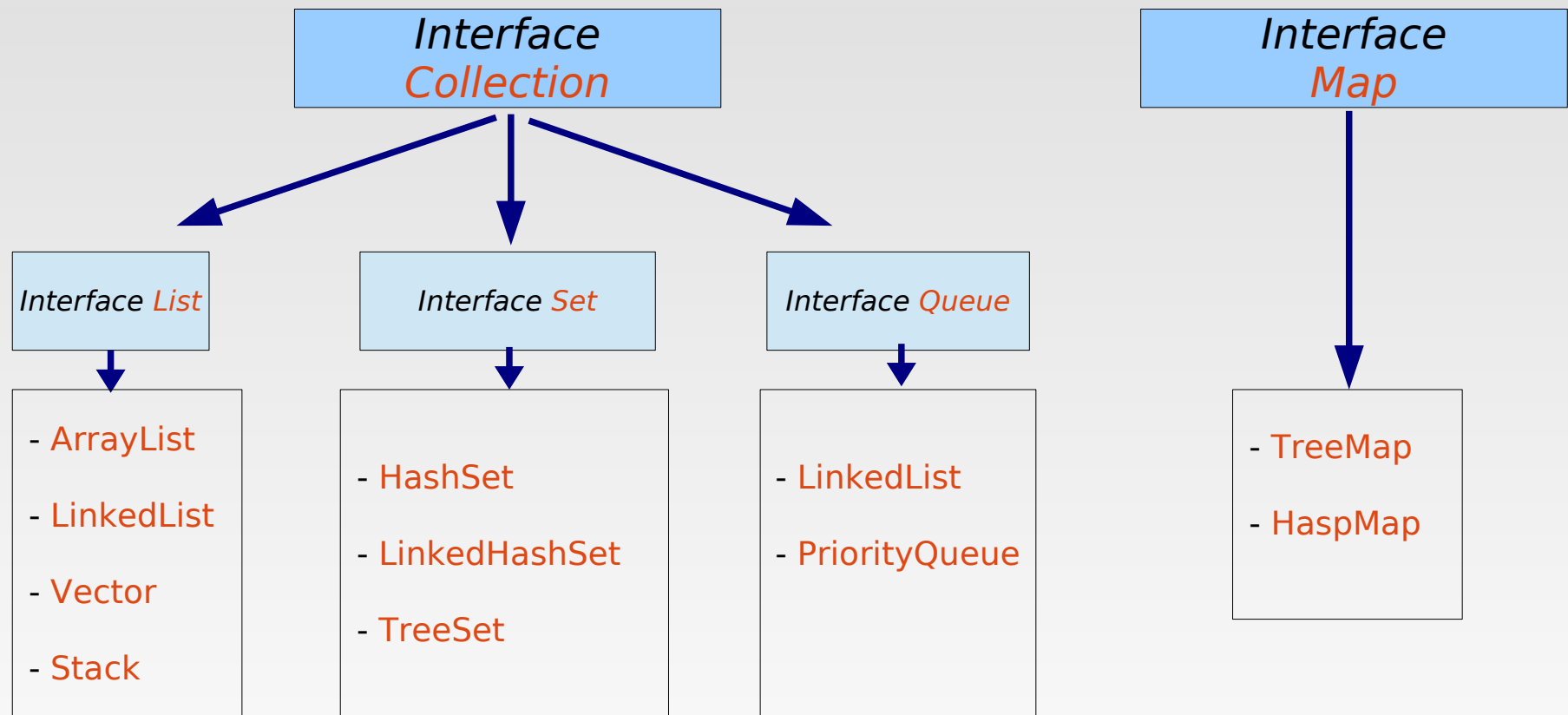


* están incluidas en el paquete `java.util`

Arrays y Strings

Más clases para controlar arrays dinámicos:

Las **clases** más importantes se organizan de la siguiente manera:



- * algunas clases implementan otros subinterfaces
- * algunas clases implementan más de un interface
- * aquí se representa un breve resumen de las clases disponibles

Arrays y Strings

Strings

- Son objetos para almacenar caracteres.
- No son lo mismo que array de caracteres: están representados por clases diferentes, y los atributos y métodos son distintos.
- Clasificación:
 - No cambian en contenido ni tamaño en tiempo de ejecución (clase `java.lang.String`)
 - Pueden cambiar en contenido o tamaño en tiempo de ejecución (clase `java.lang.StringBuffer`)

Arrays y Strings

Strings – clase `java.lang.String`

- Creación de objetos String:

- Mediante un literal: cuando el compilador encuentra un literal entre “ doble crea un objeto de la clase String.

```
String <nombreString>="Cadena";
```

o bien:

```
String <nombreString>;  
<nombreString>="Cadena"; //declarado antes
```

- Llamando a algún constructor (realmente se inicializa un apuntador a la cadena creada):

```
String <nombreString>=new String ("Cadena");
```

- Llamando a un método que devuelva un objeto String nuevo, ej:

```
String s1="Hola"; //Crea un objeto referenciado en s1  
String s2; //Crea una referencia pero no un objeto  
s2=s1.toUpperCase(); //Crea un objeto referenciado por s2
```

Arrays y Strings

Strings – clase `java.lang.String`

- Realmente, en memoria sólo existe un objeto `String` por cada cadena existente en tiempo de ejecución, a pesar de que se emplee `new`. Ej:

```
String s1=new String("Hola");  
String s2=new String("Hola");  
String s3=new String("Hola");  
System.out.println (s1.equals(s2)); //true  
System.out.println (s1.equals(s3)); //true
```

- Los literales entre “ doble no asignados a variables `String` ocupan memoria que va a ser liberada por el recolector de basura. Ej:

```
String s="Hola";  
s="Mundo"; //"Hola" es un objeto String desechable
```

Arrays y Strings

Strings – clase `java.lang.String`

- Cuando se usa el operador `+` y algún sumando es un objeto `String`:
 - se obtiene el `String` asociado al resto de sumandos (si no son objetos `String`)
 - y se devuelve un nuevo objeto `String` concatenación de todos los sumandos.
- Métodos de obtención de `String` asociado a valores de tipo primitivo y objetos de tipo referencia:
 - Para tipos primitivos: en la clase `java.lang.String`
`static String valueOf (<valorTipoPrimitivo>)`
 - Para tipos referencia: en la clase `java.lang.Object`
`String toString()`

Arrays y Strings

Strings – clase `java.lang.String`

- Métodos de interés:
 - Constructores: `String()`, `String(char array[])`, `String (String cadena)`, `String (StringBuffer buffer)`
 - Subcadenas: `charAt`, `substring`
 - Comparaciones: `compareTo`, `endsWith`, `equals`, `equalsIgnoreCase`, `startsWith`
 - Concatenar: `concat`
 - Búsquedas: `IndexOf`, `lastIndexOf`
 - Reemplazar: `replace`, `toLowerCase`, `toUpperCase`, `trim`
 - Otros: `length`, `toCharArray`, `valueOf`



Java Packages

Paquetes

Introducción

- Concepto: conjunto de clases e interfaces que se agrupan por razones de organización y para evitar problemas de nomenclatura o identificadores repetidos. Su uso es más conveniente cuanto mayor sea el proyecto.
- Equivale al concepto de librería en C.
- Se identifican con nombres formados por palabras (por convenio en minúsculas) separadas por puntos. Ej:
`java.applet`
- Se pueden anidar formando jerarquías, ej:
 - `java.awt`: paquete general del interface gráfico awt
 - `java.awt.color`, `java.awt.image`,...: otros paquetes de la jerarquía

Paquetes

Paquetes, Directorios y Ficheros .class

- Un paquete consta de uno o más ficheros .class que han de estar en un único directorio. Tal directorio se corresponde con todas las palabras del nombre del paquete, cambiando . por / Ej. los ficheros .class del paquete de nombre <dir1>.<dir2> estarían en el directorio <loquesea>/<dir1>/<dir2>/
- Los ficheros .class de un directorio no pueden pertenecer a más de un paquete, salvo al paquete por defecto (ver apartado).
- Cuando en un directorio, además de ficheros .class, hay otros subdirectorios con más ficheros .class que se corresponden a otro paquete, se dice que los paquetes están anidados.
- Por sencillez se suelen dejar los ficheros fuente en el mismo directorio que los ficheros .class, pero no es obligatorio.

Paquetes

Declaración

- Para indicar el paquete al que pertenece un fichero fuente, la primera línea del fuente será:

```
package [<dir1>.] [<dir2>.] ... <dirN>;
```

siendo dir1/dir2/.../dirN un subdirectorio en el que se vaya a guardar su fichero .class.

- Los ficheros fuente que formen el paquete serán todos los que empiecen por la misma primera línea, y tales que sus ficheros .class se guarden en el directorio asociado.

Paquetes

Uso e importación – Condiciones

- Sólo se pueden usar las clases e interfaces de otros paquetes que sean públicas.
- Para acceder a los atributos y métodos de las clases (públicas) importadas de otros paquetes han de cumplirse las restricciones de ámbito, siendo obligatoria una de estas posibilidades para los atributos y métodos a los que se accede:
 - Que sean public.
 - Que sean protected, y la clase desde la que se accede sea descendiente de la clase a la que se accede.

Paquetes

Uso e importación – Pasos

- Sentencias de importación: en el fuente donde se usen, importar las clases e interfaces=indicar a qué paquete(s) pertenecen.
- Sentencias de uso.
- Variable de entorno CLASSPATH:
 - Indica los directorios de los ficheros .class correspondientes a los paquete a emplear.
 - Es obligatorio tanto en compilación como en ejecución. Ej:
 - Paquete a1.b1.c1 en directorio /home/luz/a1/b1/c1
 - Paquete a2.b2.c2 en directorio /home/luz/a2/b2/c2
 - Paquete d3.e3 en directorio /usr/local/java/d3/e3
 - Hacer: `export CLASSPATH=/home/luz:/usr/local/java`

Paquetes

Uso e importación – Importación

- Sintaxis para importación: sentencia `import`
 - Se indica al principio del fuente después de la sentencia `package` (en caso de que ésta exista) mediante una de estas formas:
 - Se importan todas las clases e interfaces del paquete:
`import <dir1>.<dir2>...<dirN>.*;`
 - Se importan sólo las específicas:
`import <dir1>.<dir2>...<dirN>.ClasePub1;`
`import <dir1>.<dir2>...<dirN>.InterfacePub2;`
 - Para el caso de paquetes anidados sólo importa las clases e interfaces del directorio al que hace referencia, y no los que estén en subdirectorios
- Al compilar una clase que importa otra, si no encuentra el `.class` de la clase a importar compila ésta automáticamente.

Paquetes

Uso e importación – Uso

- Una vez hecha la importación, para usar la clase o interface importada hay dos posibilidades:

- Indicar el nombre de la clase importada
- Anteponer al nombre de la clase importada el nombre del paquete al que pertenece. Esto es necesario si en distintos paquetes hay coincidencia de nombres de clases. Ej:

```
import <dir1>.<dir2&gt.*;  
import <dir3>.<dir4>.ClasePub1;  
...  
<dir1>.<dir2>.ClasePub1 a;  
<dir3>.<dir4>.ClasePub1 b;
```

- Para ejecutar el main de una clase no perteneciente al paquete por defecto:
`java nombrePaquete.NombreClasePublica`

Paquetes

Paquete por defecto

- Es el paquete formado por todos los fuentes de un directorio que no tienen declaración `package`
- Acceso a las clases del paquete por defecto:
 - Sólo puede hacerse desde clases del paquete por defecto. Desde otros paquetes no es posible, aunque se acceda a clases públicas.
 - Las clases del paquete por defecto pueden acceder a otras clases de dicho paquete sin necesidad de usar `import`.
 - Si desde el paquete por defecto se usan clases de dicho paquete, para que no haya error al compilar hay dos posibilidades:
 - Que la variable `CLASSPATH` no esté definida
 - Que la variable `CLASSPATH` incluya al directorio actual, ej:
`export CLASSPATH=./:/home/luz:/usr/local/java`
- Su uso es para aplicaciones pequeñas.

Paquetes

Paquetes incluidos automáticamente

- No son necesarios:
 - Hacer un import del paquete `java.lang`
 - Incluir en el CLASSPATH el archivo `rt.jar` que contiene los paquetes y ficheros `.class` necesarios para la ejecución. (En las primeras versiones de java sí era necesario)

Paquetes

Paquetes de la especificación Sun de java

- Los paquetes de la especificación contienen las clases e interfaces empleados durante el tiempo de ejecución por la JVM.

Versión	Nº clases	Nº paquetes
Java 1.0	212	8
Java 1.1	504	23
Java 2 – 1.2	1520	59
Java 2 – 1.3	1842	76
Java 2 – 1.4	2991	135
Java 2 – 1.5	3562	166
Java 2 – 1.6		202

- Ver <http://java.sun.com/javase/6/docs/api/>

Paquetes

Paquetes de la especificación Sun de java

- `java.lang`: clases propias del lenguaje.
 - Define la clase `Object` y las clases compatibles con tipos primitivos: `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean`.
Respecto a éstas:
 - Su utilidad es para homogeneizar el tratamiento de tipos primitivos con el de clases y objetos.
 - Son más lentas que si se trabaja con tipos primitivos, y no son válidas con operadores (+ - * /...), pero tienen más funcionalidades (atributos y métodos).
 - Tienen un atributo privado para guardar el valor del tipo primitivo.
 - Es posible convertir en ambos sentidos, ej:

```
Integer ii1=new Integer (i1)
int i2 = ii2.intValue()
```
 - Es el único paquete que se importa automáticamente aunque no se indique con `import`

Paquetes

Paquetes de la especificación Sun de java

- java.io: clases para entrada/salida de datos y acceso a ficheros.
- java.util: arrays dinámicos, fechas,...
- java.util.zip: compresión zip y gzip de ficheros
- java.applet: herramientas para applets.
- java.awt y java.swing: componentes para GUIs (Graphic User interface).
- java.net: funciones de red.
- java.sql: acceso a bases de datos sql
- java.security: encriptación
- ...

Paquetes

(Anexo: formato de paquetes jar)

- Los paquetes (directorios + ficheros .class) suelen “paquetizarse” en el formato comprimido jar.

Ej. para ver los ficheros .class del API de Sun:

```
unzip -l /usr/lib/jvm/java-1.5.0-sun/jre/lib/rt.jar
```

- Si algunos paquetes están en formato jar y se quieren importar, hay que indicarlo en el CLASSPATH.

Ej.

```
export CLASSPATH=./usr/share/java/qtjava.jar:/dir/p.jar
```

- A veces se incluye la clase ejecutable en un fichero .jar y para lanzar la aplicación se emplea:

```
jar -j fichero.jar
```

(Nota: en este caso se ignora la variable CLASSPATH y para indicar otras clases y .jar se debe emplear un fichero manifest)

Paquetes

(Anexo: fichero manifest)

- Es un fichero de texto empleado cuando la clase ejecutable está en un fichero jar. El formato tiene, entre otras líneas:

```
Main-Class: paquete.ClasePrincipal  
Class-Path: dir1/jar1.jar dir2/jar2.jar dir3/ dir4/...
```

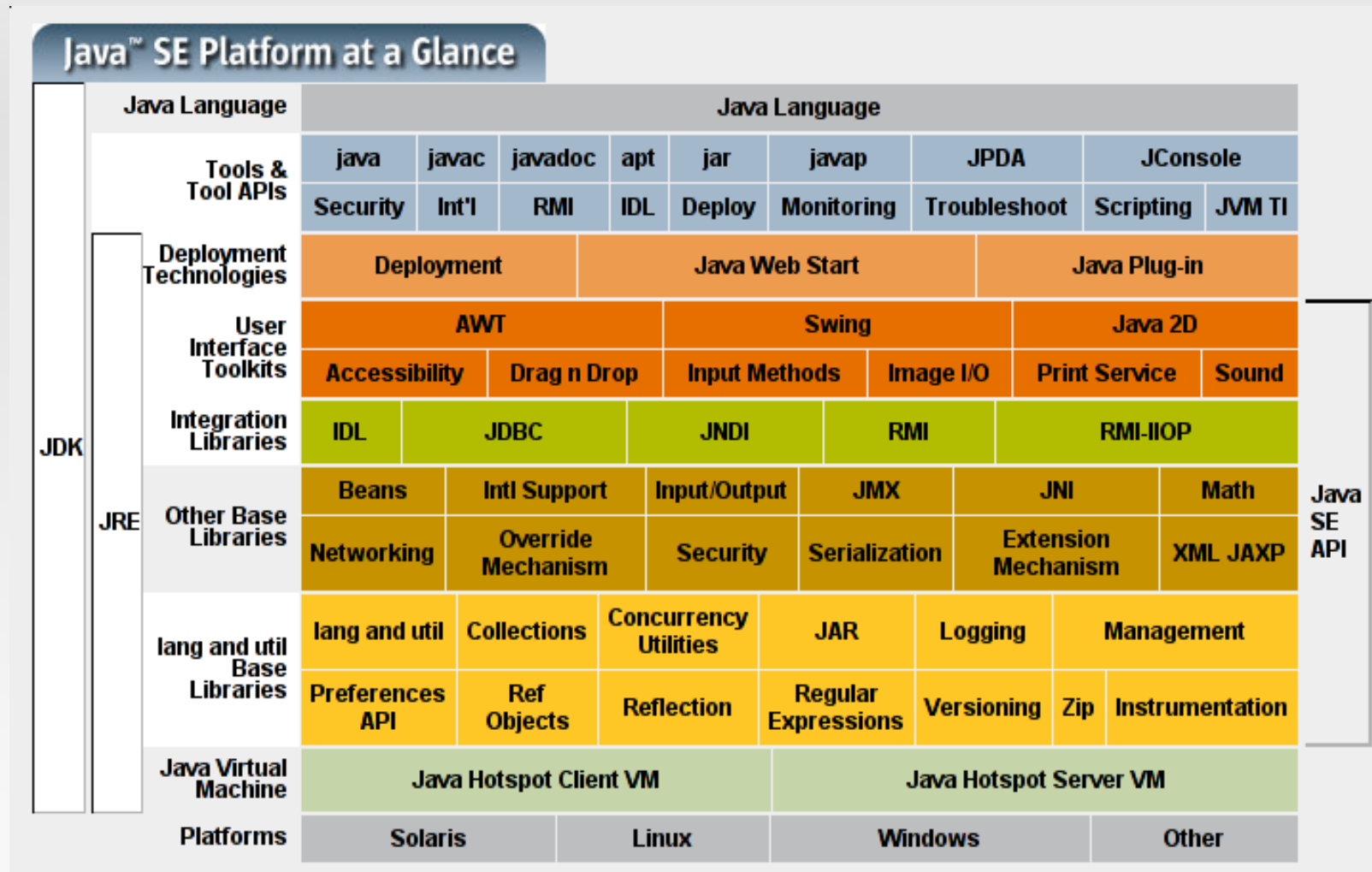
Siendo:

- `paquete.ClasePrincipal` la clase que contiene el método `main` para ejecutarse, que pertenece a uno de los paquetes que forman el jar; en caso de que sea el default, sería:
 Main-Class: ClasePrincipal
- `dirN/jarN.jar dirM/...` rutas absolutas o relativas (a la ubicación del paquete jar) de otros ficheros jar o de directorios que contengan clases, todos ellos que estén FUERA del paquete jar, ej. en la red local.
- Importante: salto de línea al final



Java API

API



<http://docs.oracle.com/javase/6/docs/api/>

<http://docs.oracle.com/javase/1.5.0/docs/api/overview-summary.html>

API – paquetes básicos

- **java.lang**: clases básicas de Java: Numbers, Math, String, Process, Thread
- **java.io**: entrada y salida de sistema a través de data streams, serialización y ficheros.
- **java.awt**: una interface de usuario, gráficos e imágenes: [más info](#)
- **java.util**: utilidades de colecciones, fechas, horas internacionalización, y utilidades varias como números aleatorios.
- **java.sql**: permite el acceso a bases de datos relacionales
- **java.net**: servicios de red (Sockets, URL, ...)
- **java.rmi**: permite realizar procesos distribuidos utilizando RMI (Remote Method Invocation)
- **java.sound**: control de sonido e interface MIDI (varios paquetes): [ejemplo](#)

API - Manejo de tipos primitivos

- Clase **abstracta** **Number**
- Heredan de ella:

AtomicInteger, AtomicLong, BigDecimal, BigInteger, Byte, **Double**, **Float**, **Integer**, Long, Short

- Permiten convertir tipos de datos a byte, double, float, int, long, and short.
- Por ejemplo:

```
int dato_int = Integer.parseInt("123");
```

convierte el string "123" en un entero con valor 123.

API - Fecha hora

- Clase `java.util.Calendar`
- Ayuda a gestionar fechas y horas .
- Por ejemplo:

```
import java.text.SimpleDateFormat;
import java.util.Calendar;

public class currentTime {
    public static void main(String[] args) {
        Calendar cal = Calendar.getInstance();

        SimpleDateFormat sdf = new SimpleDateFormat("HH:mm:ss");
        System.out.println( sdf.format(cal.getTime()) );
    }
}
```

Obtiene un calendario con los ajustes locales de zona horaria y demás.

Extrae la fecha

API - Timer

- Clase `java.util.Timer`
- Permite crear un temporizador en “background” y crear un evento cuando vence un periodo de tiempo .
- Por ejemplo:

```
import java.util.Timer;
import java.util.TimerTask;

public class Recordatorio {

    Timer timer;

    public Recordatorio(int seconds) {
        timer = new Timer();
        timer.schedule(new TareaTimer(), seconds*1000);
    }

}

Public class Aplicacion {

    public static void main(String args[]) {
        new Recordatorio(5);
        System.out.println("Tarea programada");
        Thread.sleep(6000);
    }

}
```

```
class TareaTimer extends TimerTask {

    public void run() {
        System.out.println("Se ha alcanzado el tiempo");
        timer.cancel(); //Se cancela el timer thread
    }

}
```



Java Excepciones

Excepciones

Introducción

- Evento excepcional que ocurre durante el programa, interrumpiendo el flujo normal de las sentencias, debido a motivos diversos:
 - Hardware: dispositivo estropeado, recursos insuficientes (memoria, disco),...
 - Software: división por cero, acceso a posiciones no permitidas de un array, lanzamiento explícito por quien programa,...
- Método tradicional de tratamiento de errores: comprobación de código de retorno de las funciones.
- Algunos lenguajes como Java tienen mecanismos especiales para la detección y el tratamiento de dichos eventos.

Excepciones

Clases para Excepciones

- En java una excepción se modela como una clase que descende de la clase `java.lang.Throwable`:
 - Excepciones existentes en el API de java: clases que se encuentran en distintos paquetes (`java.lang`, `java.io`, `java.util.zip`,...)
 - Excepciones creadas por el programador: nuevas clases que se deriven de la clase `Throwable`, de forma directa o indirecta.
- Métodos de interés de la clase `java.lang.Throwable`:
 - `public String getMessage()`: mensaje de error asociado
 - `public String toString()`: descripción breve del objeto
 - `public String printStackTrace()`: imprime una traza de la pila

Excepciones

Clases para Excepciones

- Árbol del paquete java.lang:

<http://java.sun.com/javase/6/docs/api/java/lang/package-tree.html>

java.lang.Object

java.lang.Throwable (implements java.io.Serializable)

└─ java.lang.Error

...

└─ java.lang.Exception

 java.lang.RuntimeException

...

- Clase **java.lang.Error**: de uso propio de la JVM. Son errores irre recuperables y se finaliza el programa.
- Clase **java.lang.Exception**: son errores que el programa podría capturar y tratar.

Excepciones

Tipos de excepciones

- No comprobables o implícitas (“unchecked”):
 - No se comprueban durante la fase de compilación.
 - Son las clases:
 - `java.lang.Error` y derivadas
 - `java.lang.RuntimeException` y derivadas, ej: punteros null, índice fuera de array,...
- Comprobables o explícitas (“checked”):
 - Se comprueban durante la fase de compilación, y es obligatorio tratarlas o declararlas en el método.
 - Son las derivadas de `java.lang.Exception` quitando la clase `java.lang.RuntimeException` y sus subclases. Ejs: `IOException`, `CloneNotSupportedException`

Excepciones

Lanzamiento de una excepción

- Durante la ejecución de un método se pueden producir o “lanzar” (throw) excepciones de dos formas:
 - La JVM detecta una situación especial, Hardware (ej. un error al cargar una parte del programa) o Software (ej. división por cero).
 - En un método se lanza una excepción por código mediante throw:
`throw <objetoDeClaseExcepcion>`
Ej: `throw new Exception();`
- En ambos casos se emplea un objeto de la clase correspondiente a la excepción (en el primer caso lo crea la JVM).

Excepciones

Tratamiento de excepciones

- Las excepciones, tanto si son explícitas como implícitas, pueden ser tratadas en el mismo método mediante estructuras de lenguaje java llamadas manejadores de excepciones.
- Un manejador de excepciones consiste en:
 - Una bloque de código “try” que contiene el código que puede generar excepciones.
 - Uno o más bloques “catch” donde se realiza el tratamiento de las excepciones que pueden producirse.
 - Un bloque opcional “finally” en el que se incluye código final a ejecutar siempre. Utilidad: para no repetir código en todos los bloques catch. Ej. cierre de ficheros si hay o no errores.

Excepciones

Tratamiento de excepciones – Sintaxis

```
try {  
    <cuerpo del try>  
}  
catch (<ClaseExcepcion1> <param1>) {  
    <cuerpo del catch 1>  
}  
[catch (<ClaseExcepcion2> <param2>) {  
    <cuerpo del catch 2>  
}]  
...  
[finally {  
    <cuerpo del finally>  
}]
```

- Entre los distintos bloques try, catch y finally no puede haber otras líneas de código.
- Dentro de los bloques catch y finally también podría haber nuevos bloques try-catch-finally (gestión recursiva).

Excepciones

Tratamiento de excepciones – Bloques catch

- Suele existir al menos un bloque catch. No es obligatorio pero es lo usual; si no hay ninguno ha de haber un finally.
- Si hay varios bloques catch han de tratar excepciones distintas.
- Se produce un error de compilación si hay varios bloques catch que tratan excepciones relacionadas por herencia y se escribe primero el catch de las excepciones más generales (antepasadas) y después el de las concretas (hijas).

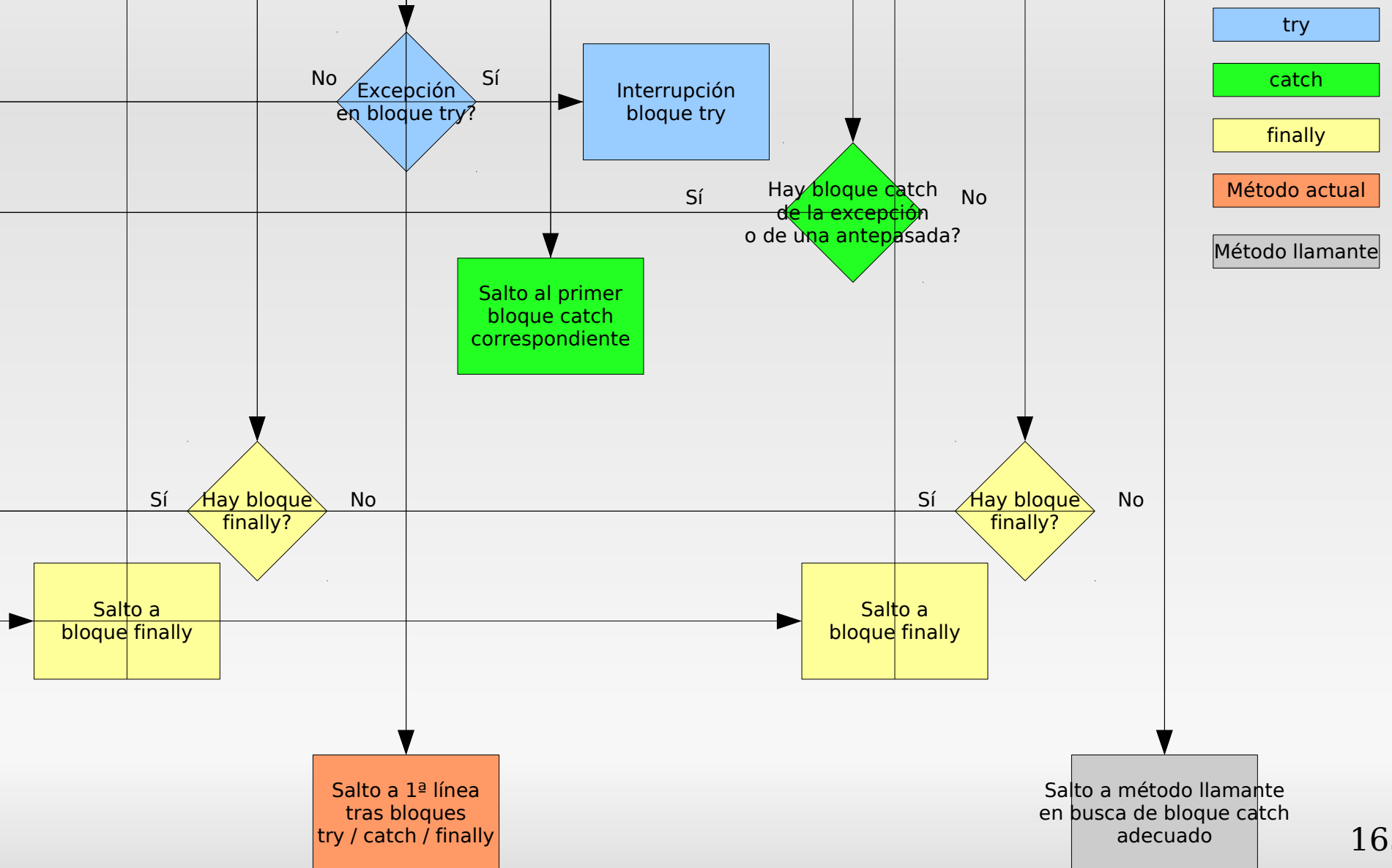
Excepciones

Tratamiento de excepciones – Secuencia

- Cuando hay una excepción en un método, el flujo se interrumpe y la JVM mira si el método en el que se ha detectado la excepción puede tratarla mediante un bloque catch para esa excepción o una antepasada de ésta. (Ver diagrama de flujo).
- En caso negativo, pasa por el bloque finally si existe, y después recorre la cadena de métodos desde los que se llamó, buscando algún catch que pueda tratar la excepción:
 - En el caso de programas standalone llega hasta el método main
 - Si no encuentra ninguno que pueda tratarlo, la JVM muestra un mensaje de error y el programa termina.

Excepciones

Secuencia de una excepción – Diagrama de Flujo



Excepciones

Excepciones explícitas – Uso de throws (i)

- Sintaxis de declaración de métodos:

`...<nombreMétodo> ([<parámetros>]) throws <ListaExcepciones>`

Lista de excepciones: separadas por comas

- Se usa (y es obligatorio) para indicar aquellas excepciones explícitas que se pueden generar dentro del método, pero en dicho método no van a ser tratadas ni ellas ni sus superclases, sino que se espera que se traten en métodos llamantes.
- En un método se pueden producir excepciones explícitas mediante throw.

Excepciones

Excepciones explícitas – Tratamiento

- Se produce un error de compilación si en un método se pueden lanzar excepciones explícitas (con `throw`), y dicho método no cumple ninguna de estas posibilidades:
 - Tratar en algún `catch` dichas excepciones o sus ascendientes.
 - Declarar con `throws` dichas excepciones o sus ascendientes:
- ...<nombreMétodo>([<parámetros>]) throws <ExcepcionesExplicitas>

- Consecuencia: al llamar a un método es obligatorio conocer sus excepciones explícitas declaradas con `throws`, y hacer alguna(s) de las opciones indicadas. (Ver “Entrada por `stdin`”).

Excepciones

Excepciones explícitas – Tratamiento

- Se produce un error de compilación si algún bloque catch corresponde a una excepción explícita, y no se cumple que en el bloque try se pueda lanzar dicha excepción o alguna descendiente.
- Lo anterior no se aplica para la clase Exception.

Excepciones

Excepciones explícitas – Herencia

- Si en una clase se redefinen métodos que en la superclase lanzan excepciones explícitas, en la declaración de dichos métodos de la clase:
 - No se podrán indicar excepciones explícitas que no estén declaradas con throws en el método de la superclase. Sí podrán indicarse menos.
 - Salvedad a lo anterior: se pueden indicar nuevas excepciones explícitas que sean descendientes de las excepciones declaradas en el método de la superclase.
- No aplica para los constructores, pues no se heredan.

Excepciones

Excepciones explícitas – Uso de throws (ii)

- Como previsión para subclasses:
En un método de una clase se puede indicar un número suficiente de excepciones explícitas para que, aunque en dicho método no sean lanzadas, sí puedan incluirse en la declaración de los métodos reescritos de clases derivadas.

Excepciones

(Recomendaciones)

- Separar la lógica del programa y el código de tratamiento de error mediante gestores de excepciones.
- Lanzar las menos excepciones posibles, y preferentemente que sean explícitas.
- Tratamiento:
 - No dejar bloques catch de excepciones con cuerpo vacío.
 - No atrapar excepciones que vengan de `java.lang.Error` porque son propias de la JVM y pueden implicar estados inconsistentes.
 - Propagar lo menos posible las excepciones producidas en un método, intentando tratarlas cuanto antes.
 - Usar `finally` para gestión final de recursos (ej. cierre de ficheros).



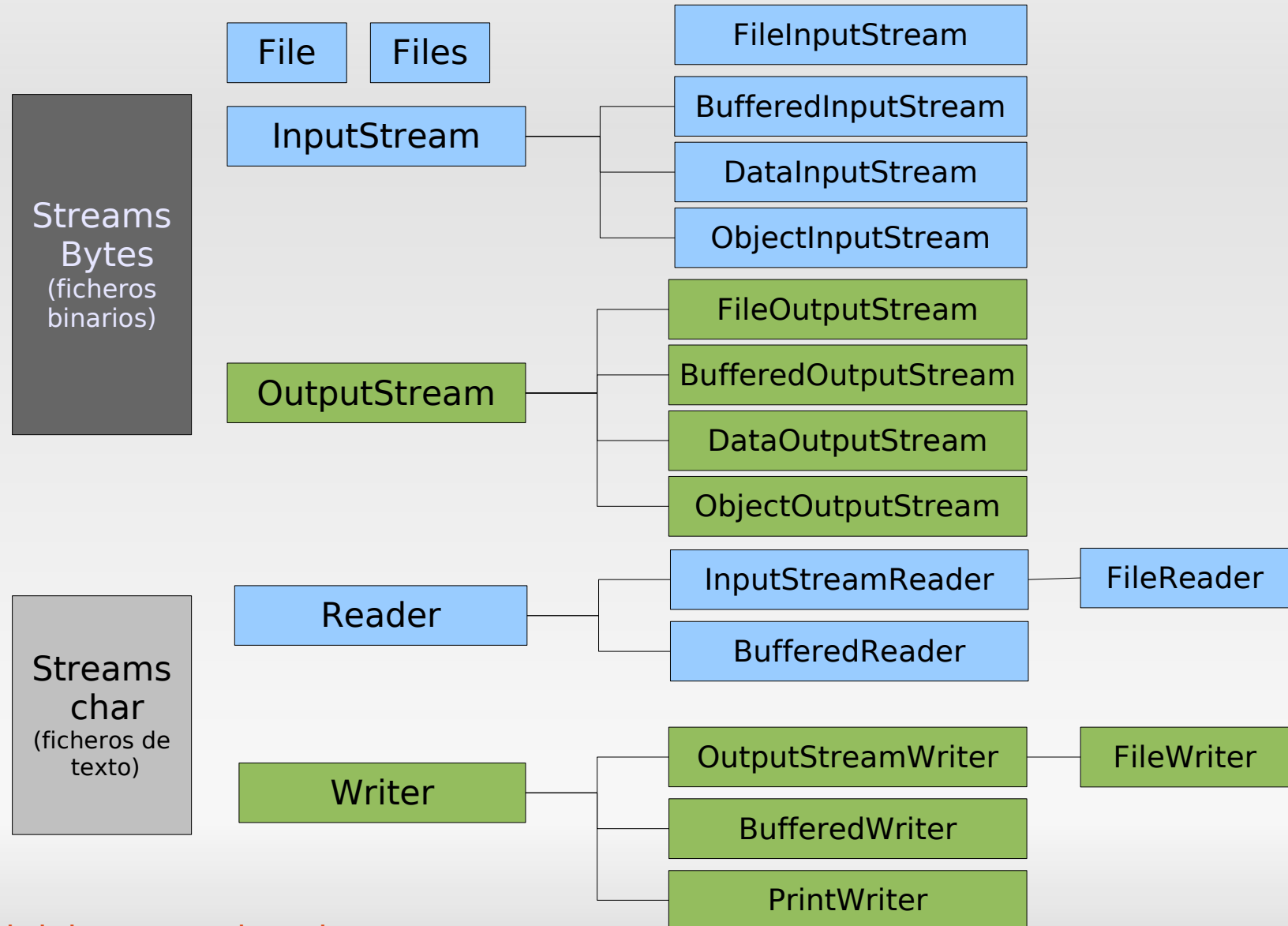
Java Ficheros

Ficheros - generalidades

- Existe una serie de clases que permite el acceso y el uso de ficheros para guardar y leer información.
- La gestión de ficheros está incluida en el paquete “**java.io**”
- Para la gestión básica del sistema de ficheros (crear un fichero, listar ficheros de un directorio, borrar un fichero, ...) existe la clase “**File**”.
- Para acceder a los datos de un fichero se crean **streams** tanto de bytes como de caracteres sobre los ficheros.
- Las clases de streams de ficheros incluyen la palabra “**File**” (FileInputStream, FileOutputStream, FileReader, FileWriter, ...).
- Una vez creado un stream de Fichero se pueden crear otros streams sobre esos ficheros para acceder de forma más cómoda y eficiente a los datos. Los “**BufferedStream**” por ejemplo reducen el número de acceso a disco ya que cuentan con un buffer intermedio transparente al usuario.

Ficheros — clases más populares

Jerarquía de clases más populares relacionadas con ficheros en java.io



Ficheros — clase File

- Permite acceder a un fichero o a un directorio de forma genérica para obtener sus propiedades, renombrarlo, borrarlo, crear un directorio, etc.

- Constructor:

`File (String pathname)`

- Métodos más importantes:

`exists(), canRead(), canWrite(), canExecute()`

`mkdir(), renameTo(), list()`

`delete()`

Ficheros — clase Files

- Contiene una colección de métodos estáticos para acceder a un fichero o un directorio. Permite realizar las acciones habituales que se realizan con un gestor de ficheros.
- Métodos más importantes:
 - `copy(...)`, `move(...)`, `delete(...)`
 - `createFile(...)`, `createDirectory(...)`
 - `isExecutable(...)`, `isReadable(...)`, `isXXX...`
 - `readAllBytes(...)`, `write(...)`

Ficheros — apertura y cierre

- Un fichero se puede abrir a través del **constructor** de una clase “xxFilexx” pasándole un objeto de tipo File o el nombre del fichero.
- En caso de apertura para escritura se puede abrir en modo “append” para añadir datos al final del fichero.
- Ejemplos:
 - `public FileInputStream(String name)`
 - `public FileOutputStream(String name, boolean append)`
 - `public FileOutputStream(File file)`
 - `public FileReader(String fileName)`
 - `Public FileWriter(String fileName)`
- Para cerrar el fichero se utiliza el método **close()**.
- Las clases de acceso a ficheros lanzan excepciones que hay que gestionarlas para conocer las causas de los errores (fichero no existe,...)

Ficheros — apertura y cierre

- Un fichero se puede abrir a través del **constructor** de una clase “xxFilexx” pasándole un objeto de tipo File o el nombre del fichero.
- En caso de apertura para escritura se puede abrir en modo “append” para añadir datos al final del fichero.
- Ejemplos:
 - `public FileInputStream(String name)`
 - `public FileOutputStream(String name, boolean append)`
 - `public FileOutputStream(File file)`
 - `public FileReader(String fileName)`
 - `Public FileWriter(String fileName)`
- Para cerrar el fichero se utiliza el método **close()**.
- Las clases de acceso a ficheros lanzan excepciones que hay que gestionarlas para conocer las causas de los errores (fichero no existe,...)

Ficheros – binarios

- El acceso a un fichero binario se puede realizar a través de las clases
 - “FileInputStream” para el caso de lectura.
 - “FileOutputStream” para el caso de escritura.

Estas clases nos ofrecen un acceso “directo” a los datos del fichero pudiendo hacer escritura y lectura “cruda” de conjuntos de bytes.

- Una vez obtenido el acceso al fichero se pueden crear nuevos stream
 - para realizar el acceso de forma “bufferizada”:
 - InputStream, OutputStream
 - para escribir y leer datos primitivos:
 - DataInputStream, DataOutputStream
 - o incluso objetos:
 - ObjectInputStream, ObjectOutputStream

Ficheros binarios - FileInputStream

Ejemplo mínimo de FileInputStream

```
public static void main(String[] args) {  
  
    try {  
  
        byte DatosLeidos[]=new byte[3];  
        int NumBytesLeidos;  
  
        FileInputStream miFileInputStream=new FileInputStream("<nombre de fichero>");  
  
        NumBytesLeidos = miFileInputStream.read(DatosLeidos);  
  
        System.out.println("Leidos " + NumBytesLeidos + " del fichero: " + DatosLeidos[0] + " "  
+ DatosLeidos[1] + " " + DatosLeidos[2] );  
  
        miFileInputStream.close();  
  
    }  
    catch (FileNotFoundException e) {  
        System.out.println("El fichero no existe");  
    }  
    catch(Exception e) {  
        System.out.println("Excepción:" + e.getMessage());  
    }  
}
```

Ficheros binarios - FileOutputStream

Ejemplo mínimo de FileOutputStream

```
public static void main(String[] args) {  
    try {  
        byte DatoEscribir[]=new byte[3];  
  
        DatoEscribir[0]=1;  
        DatoEscribir[1]=2;  
        DatoEscribir[2]=3;  
  
        FileOutputStream miFileOutputStream=new FileOutputStream("<nombre de fichero>");  
  
        miFileOutputStream.write(DatoEscribir);  
  
        miFileOutputStream.close();  
    }  
    catch (FileNotFoundException e) {  
        System.out.println("El fichero no existe");  
    }  
    catch(Exception e) {  
        System.out.println("Excepción:" + e.getMessage());  
    }  
}
```

Ficheros binarios - DataInputStream

Ejemplo mínimo de DataInputStream

```
public static void main(String[] args) {  
  
    int DatoInt;  
  
    try {  
  
        FileInputStream miFileInputStream=new FileInputStream("<nombre de fichero>");  
        DataInputStream miDataInputStream=new DataInputStream(miFileInputStream);  
  
        DatoInt=miDataInputStream.readInt();  
  
        System.out.println("Leido el dato int: " + DatoInt);  
  
        miDataInputStream.close();  
        miFileInputStream.close();  
  
    }  
    catch (FileNotFoundException e) {  
        System.out.println("El fichero no existe");  
    }  
    catch (Exception e) {  
        System.out.println("Excepción:" + e.getMessage());  
    }  
}
```

Ficheros binarios - DataOutputStream

Ejemplo mínimo de DataOutputStream

```
public static void main(String[] args) {  
  
    try {  
  
        int DatoInt = 123;  
  
        FileOutputStream miFileOutputStream = new FileOutputStream("<nombre de fichero>");  
        DataOutputStream miDataOutputStream = new DataOutputStream(miFileOutputStream);  
  
        miDataOutputStream.writeInt(DatoInt);  
  
        miDataOutputStream.close();  
        miFileOutputStream.close();  
  
    }  
    catch (FileNotFoundException e) {  
        System.out.println("El fichero no existe");  
    }  
    catch (Exception e) {  
        System.out.println("Excepción:" + e.getMessage());  
    }  
}
```

Ficheros binarios - ObjectOutputStream

Para poder guardar un objeto en fichero debe implementar la interfaz "Serializable".

```
public class ClaseA implements Serializable {

    int x;
    char c;
    float f;
    String Texto;

    public ClaseA(int _x, char _c, float _f, String _texto)
    {
        x=_x;
        c=_c;
        f=_f;
        Texto=_texto;
    }

    @Override
    public String toString()
    {
        return "x=" + x + " / c=" + c + " / f=" + f + " / Texto=" + Texto;
    }

}
```


Ficheros binarios - ObjectOutputStream

Ejemplo mínimo de ObjectOutputStream

```
public static void LeerDeFichero()
{
    try {

        ClaseA miclaseA;

        FileInputStream miFileInputStream = new FileInputStream("<fichero imagen>");
        ObjectOutputStream miObjectInputStream = new ObjectOutputStream(miFileInputStream);

        miclaseA=(ClaseA) miObjectInputStream.readObject();

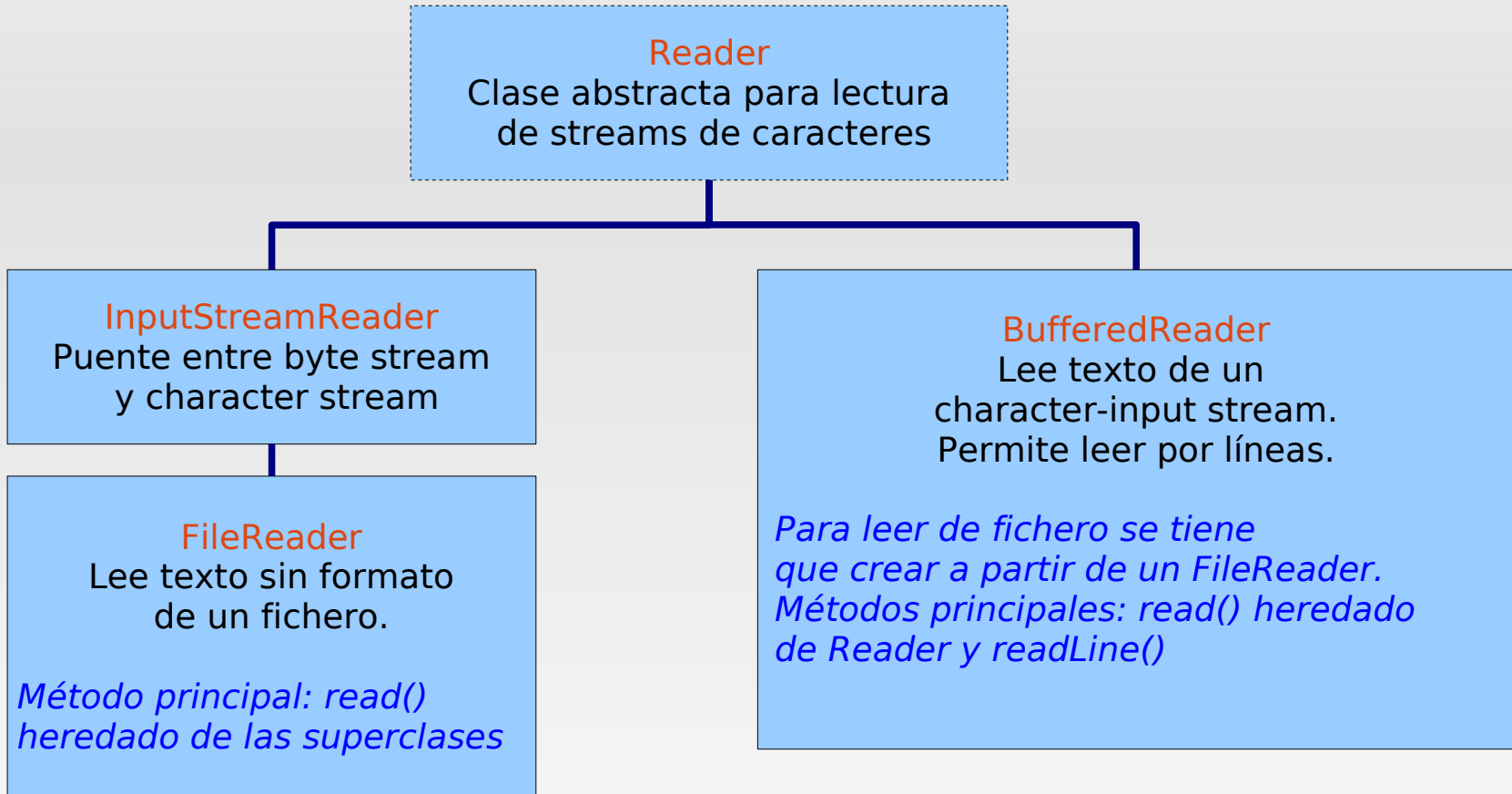
        System.out.println("Leido objeto clase A: " + miclaseA.toString());

        miObjectInputStream.close();
        miFileInputStream.close();

    }
    catch (FileNotFoundException e) {
        System.out.println("El fichero no existe");
    }
    catch (ClassNotFoundException e) {
        System.out.println("Excepción Class Not Found:" + e.getMessage());
    }
    catch (IOException e) {
        System.out.println("Excepción I/O:" + e.getMessage());
    }
    catch (Exception e) {
        System.out.println("Excepción:" + e.getMessage());
    }
}
```

Ficheros de texto – Reader 1

Clases más importantes heredadas de Reader



```
int read(char[] cbuf) throws IOException
int read(char[] cbuf, int offset, int length) throws IOException
String readLine() throws IOException
```

Ficheros de texto – Reader 2

Ejemplo mínimo de FileReader

```
public static void main(String[] args) {  
  
    char BufferLectura[]=new char[1000];  
    int NumCharsLeidos;  
    String StringLeido;  
  
    try {  
        FileReader miFileReader=new  
FileReader("/home/jtpaberg/EjemploFicheros/FichTexto.txt");  
  
        NumCharsLeidos=miFileReader.read(BufferLectura);  
        StringLeido=new String(BufferLectura);  
        StringLeido=StringLeido.trim();  
  
        System.out.println("Se han leído " + NumCharsLeidos + " caracteres.");  
        System.out.println(" Leído: '" + StringLeido + "'");  
  
        miFileReader.close();  
  
    }  
    catch (FileNotFoundException e) {  
        System.out.println("El fichero no existe");  
    }  
    catch(Exception e) {  
        System.out.println("Excepción:" + e.getMessage());  
    }  
}
```

Las opciones de lectura de
FileReader se limitan a leer
un buffer de caracteres.

Ficheros de texto – Reader 3

Ejemplo mínimo de BufferedReader

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

    String StringLeido;
    BufferedReader miBufferedReader;
    int NumLineas;
    int i;

    try {
        FileReader miFileReader=new FileReader("/home/jtpaberg/EjemploFicheros/FichTexto.txt");

        miBufferedReader=new BufferedReader(miFileReader);

        for (i=0;i<3;i++) {
            StringLeido=miBufferedReader.readLine();

            System.out.print("Línea [" + i + "]: ");
            System.out.println(StringLeido);
        }
        miBufferedReader.close();
        miFileReader.close();
    }
    catch (FileNotFoundException e) {
        System.out.println("El fichero no existe");
    }
    catch (Exception e) {
        System.out.println("Excepción:" + e.getMessage());
    }
}
```

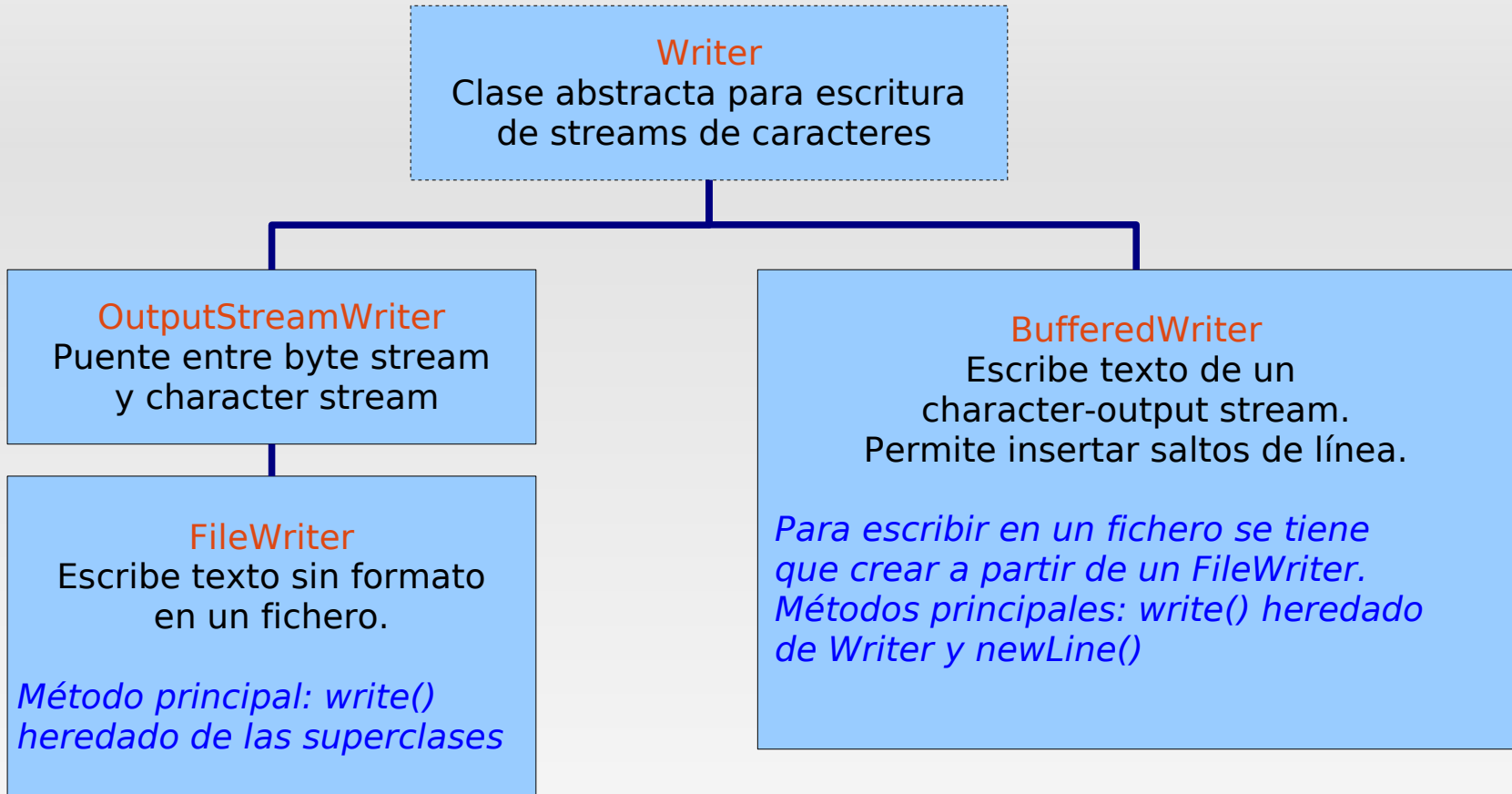
Ficheros de texto – Reader 4

Ejemplo mínimo de InputStreamReader y Buffered Reader

```
public static void main(String[] args) {  
    try {  
        String LineaLeida;  
  
        FileInputStream miFileInputStream=new FileInputStream("<nombre de fichero>");  
        InputStreamReader miISR=new InputStreamReader(miFileInputStream);  
        BufferedReader miBR=new BufferedReader(miISR);  
  
        LineaLeida=miBR.readLine();  
  
        System.out.println("LineaLeida= " + LineaLeida);  
  
        miBR.close();  
        miISR.close();  
        miFileInputStream.close();  
  
        miISR=new InputStreamReader(System.in);  
        miBR=new BufferedReader(miISR);  
  
        LineaLeida=miBR.readLine();  
  
        System.out.println("LineaLeida= " + LineaLeida);  
  
        miBR.close();  
        miISR.close();  
    }  
    catch (Exception e) {  
        System.out.println("Excepción: " + e.getMessage());  
    }  
}
```

Ficheros de texto – Writer 1

Clases más importantes heredadas de Writer



```
void write(char[] cbuf) throws IOException
void write(char[] cbuf, int offset, int length) throws IOException
void write(String s) throws IOException
```

Ficheros de texto – Writer 2

Ejemplo mínimo de FileWriter

```
public static void main(String[] args) {  
  
    char BufferEscritura[]=new char[1000];  
    String StringEscrito;  
  
    try {  
        FileWriter miFileWriter=new FileWriter("/home/FichTextoWrite.txt");  
  
        StringEscrito="Texto de prueba de escritura\n";  
  
        miFileWriter.write(StringEscrito);  
  
        System.out.println("Se han escrito: '" + StringEscrito + "'");  
  
        miFileWriter.close();  
  
    }  
    catch (FileNotFoundException e) {  
        System.out.println("El fichero no existe");  
    }  
    catch (Exception e) {  
        System.out.println("Excepción:" + e.getMessage());  
    }  
}
```

Ficheros de texto – Writer 2

Ejemplo mínimo de BufferedWriter

```
public static void main(String[] args) {  
  
    char BufferEscritura[]=new char[1000];  
    String StringEscrito;  
  
    try {  
        FileWriter miFileWriter=new FileWriter("/home/FichTextoWriteBuff.txt");  
  
        BufferedWriter miBufferedWriter=new BufferedWriter(miFileWriter);  
  
        StringEscrito="Texto de prueba de escritura";  
  
        miBufferedWriter.write(StringEscrito);  
  
        System.out.println("Se han escrito: '" + StringEscrito + "'");  
  
        miBufferedWriter.close();  
        miFileWriter.close();  
  
    }  
    catch (FileNotFoundException e) {  
        System.out.println("El fichero no existe");  
    }  
    catch(Exception e) {  
        System.out.println("Excepción:" + e.getMessage());  
    }  
}
```



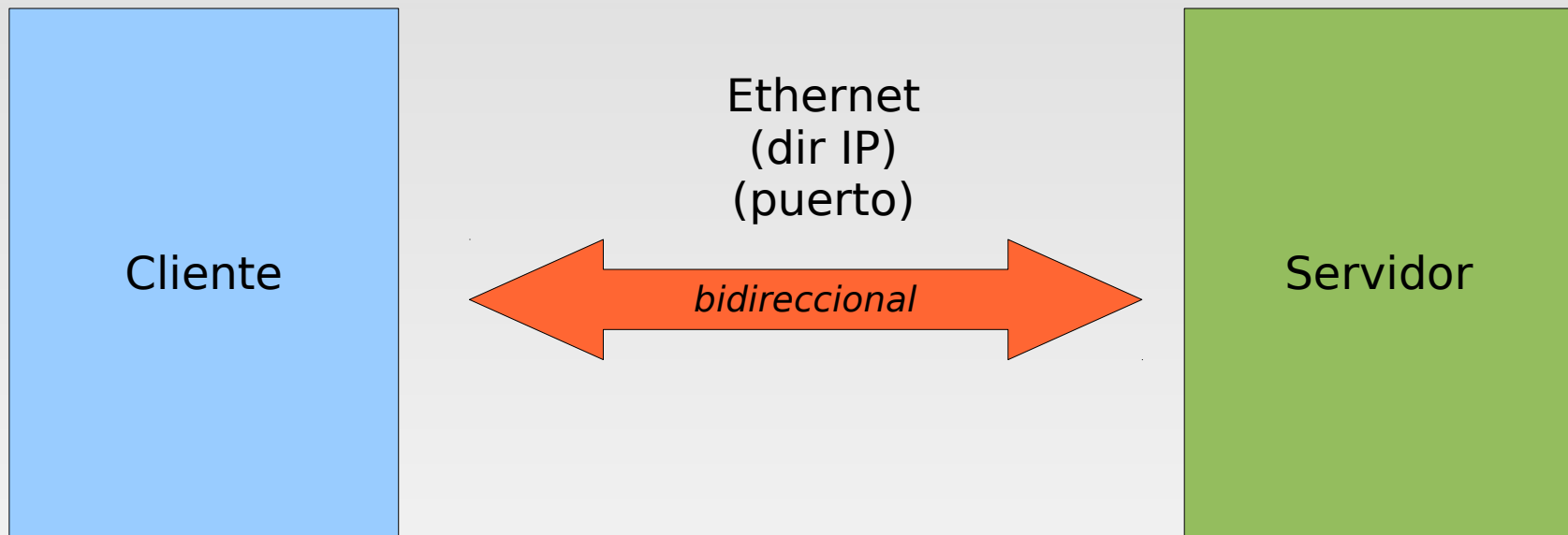

Java Sockets

Sockets - concepto

- Se manda un paquete de datos a una IP y a un puerto.

(UDP)

- Se recibe un paquete de datos en un puerto.



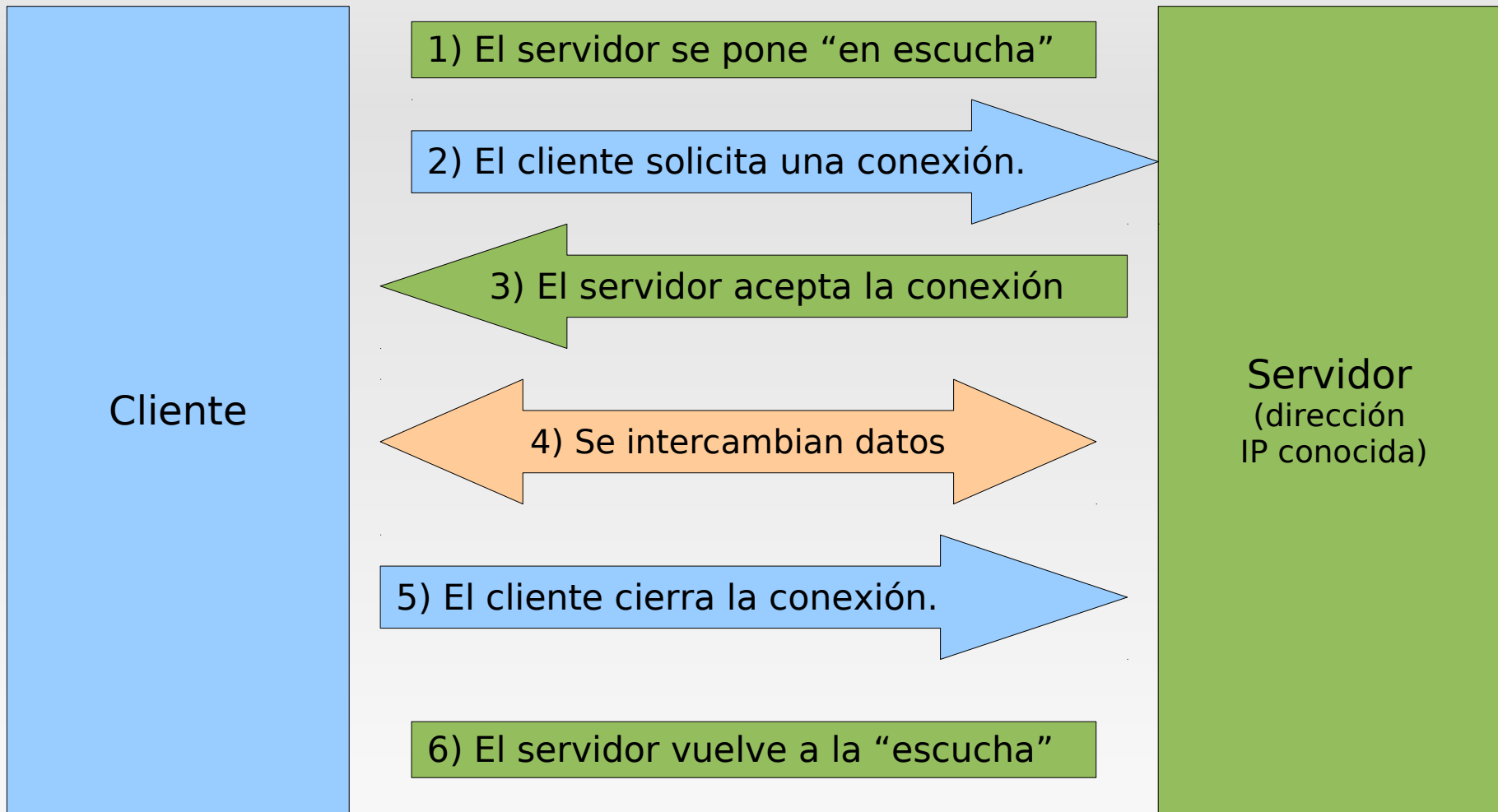
- Toma la iniciativa de conectarse o desconectarse.

(TCP/IP)

- Siempre “está ahí”.
- Si no está conectado está “en escucha” por si un cliente quiere conectarse.
- Recibe peticiones de conexión de clientes y decide aceptarlas o no.
- Cuando se conectan se pueden intercambiar datos.

Sockets – TCP

Secuencia de conexión



* El servidor también podría cerrar la conexión.

Sockets – en Java

- Las comunicaciones por red se encuentran en el paquete **java.net**.
- Las clases básicas son:
 - **InetAddress**: dirección IP.
 - **Socket**: socket cliente TCP/IP.
 - **ServerSocket**: socket servidor TCP/IP.
 - **DatagramSocket**: socket para UDP.
 - **DatagramPacket**: paquete para envío por UDP.

Sockets – clase InetAddress

- Representa una dirección IP
- Los sockets que toman la iniciativa de conectarse o enviar se pueden crear a partir de un InetAddress.
- Se pueden utilizar sus propios métodos estáticos para obtener una InetAddress:
 - InetAddress **getLocalHost()**
 - InetAddress **getByName**(String host_name)
 - InetAddress **getByAddress**(String host, byte[] addr)

Sockets – clase Socket

- Representa un socket cliente TCP/IP
- Se puede crear a través de una instancia de `InetAddress` o directamente a través de un nombre de host:
 - `Socket(InetAddress address, int port)`
 - `Socket(String host, int port)`

El constructor directamente intenta establecer la conexión con el servidor.
- Los métodos más utilizados:
 - `InputStream getInputStream()`: obtiene un stream de lectura
 - `OutputStream getOutputStream()`: obtiene un stream de escritura
 - `close()`: cierra la conexión.

Sockets – ejemplo classSocket

- Ejemplo mínimo para establecer una conexión:

```
public static void main(String[] args) {  
  
    try {  
        Socket socket = new Socket("www.ehu.es", 80);  
  
        System.out.println("Conexión establecida");  
  
        socket.close();  
    }  
  
    catch(Exception e) {  
        System.out.println("Excepción al crear el socket:  
"+e.getMessage());  
    }  
  
}
```

Sockets – ejemplo classSocket

- Un vez establecida la conexión se obtienen los stream de lectura y de escritura:

```
byte BufferLectura[]=new byte[100];
```

```
InputStream miInputStream=socket.getInputStream();
```

```
OutputStream miOutputStream=socket.getOutputStream();
```

```
// Escribir un dato
```

```
miOutputStream.write("Dato prueba".getBytes());
```

```
miOutputStream.flush();
```

```
// Leer un dato
```

```
miInputStream.read(BufferLectura);
```

```
String DatoLeido=new String(BufferLectura);
```

```
System.out.println("Leido: " + DatoLeido);
```

- * *Con el InputStream y el OutputStream el intercambio es de bytes.*
- * *Son válidas aquí las variantes de stream utilizadas en ficheros.*

Sockets – class ServerSocket

- Mediante el constructor se puede crear un server socket en un puerto determinado. Por ejemplo:

```
ServerSocket serversocket=new ServerSocket(3000);
```

- Para aceptar conexiones basta con utilizar el método accept(). Por ejemplo:

```
Socket clientSocket=serversocket.accept();
```

- Devuelve un nuevo socket que es el que se utiliza para el intercambio de datos igual que en el caso del cliente.

Sockets – ejemplo ServerSocket

```
public static void main(String[] args) {  
  
    int Puerto=3000;  
    Socket clientSocket;  
    byte BufferLectura[]=new byte[100];  
  
    try {  
        // Crear el server socket  
        ServerSocket serversocket=new ServerSocket(Puerto);  
  
        // Esperar conexión  
        System.out.println("Esperando conexión...");  
        clientSocket=serversocket.accept();  
  
        // Conexión establecida, capturar streams  
        InputStream miInputStream=clientSocket.getInputStream();  
        OutputStream miOutputStream=clientSocket.getOutputStream();  
  
        // Leer dato  
        miInputStream.read(BufferLectura);  
        String DatoLeido=new String(BufferLectura);  
  
        // Enviar eco  
        miOutputStream.write(BufferLectura);  
        miOutputStream.flush();  
  
        // Cerrar el server socket  
        serversocket.close();  
    }  
    catch (Exception e) {  
        System.out.println("Excepción al crear el server: "+e.getMessage());  
    }  
}
```

Sockets — atender a varios clientes

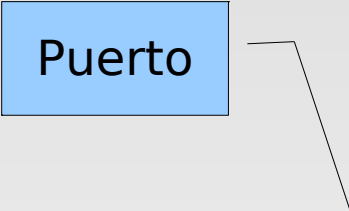
- Se puede atender a varios clientes creando un socket para cada uno y creando un thread para atender a cada uno de ellos.
- Para ellos se cuenta con la clase **Thread**

Sockets – UDP

- Para realizar un intercambio de datos bajo el protocolo UDP se cuenta con las clases **DatagramSocket** y **DatagramPacket**.
- **DatagramPacket** permite crear un paquete a enviar con una dirección IP y un puerto.
- **DatagramSocket** permite enviar ese paquete o recibirlo.

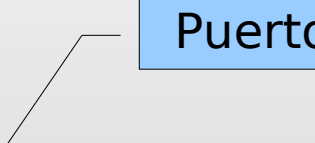
Sockets — UDP ejemplo de envío

```
public static void main(String[] args) {  
    try {  
        DatagramSocket datagramSocket = new DatagramSocket();  
        byte[] buffer = "0123456789".getBytes();  
        InetAddress receiverAddress = InetAddress.getLocalHost();  
        DatagramPacket packet = new DatagramPacket(buffer, buffer.length, receiverAddress, 3001);  
        datagramSocket.send(packet);  
        datagramSocket.close();  
    }  
    catch (Exception e){  
        System.out.println("Excepción en el envío UDP:" + e.getMessage());  
    }  
}
```



Sockets – UDP ejemplo de recepción

```
public static void main(String[] args) {  
    try {  
        DatagramSocket datagramSocket = new DatagramSocket(3001);  
  
        byte[] buffer = new byte[100];  
        DatagramPacket packet = new DatagramPacket(buffer, buffer.length);  
  
        datagramSocket.receive(packet);  
        String DatosRecibidos=new String(buffer);  
        System.out.println("Recibido: "+ DatosRecibidos);  
  
        datagramSocket.close();  
    }  
  
    catch (Exception e){  
        System.out.println("Excepción en la recepción UDP:" + e.getMessage());  
    }  
}
```



Puerto



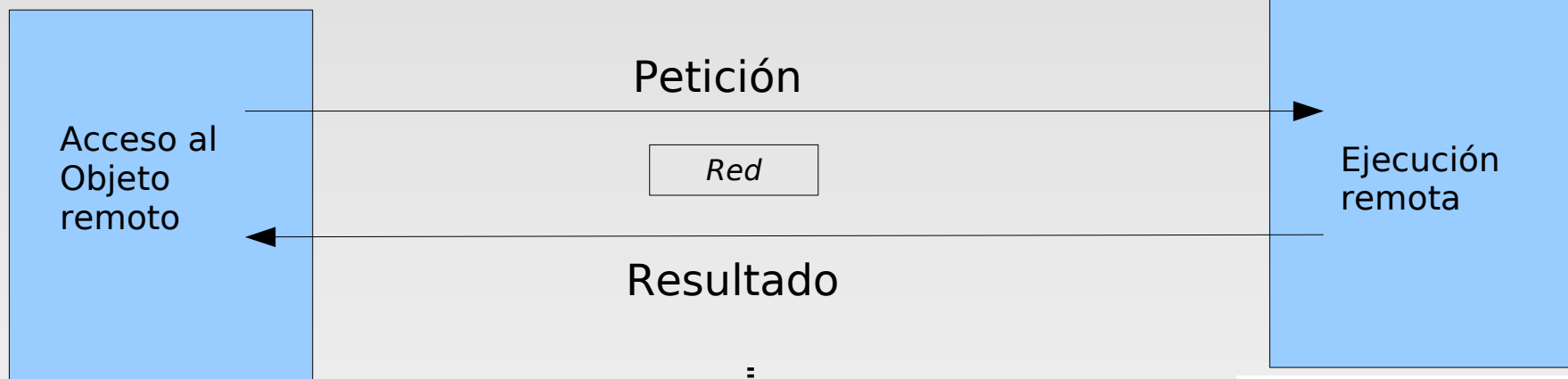
Java RMI

RMI – Remote Method Invocation

Concepto

Maquina virtual de Java (cliente)

Maquina virtual de Java (servidor)



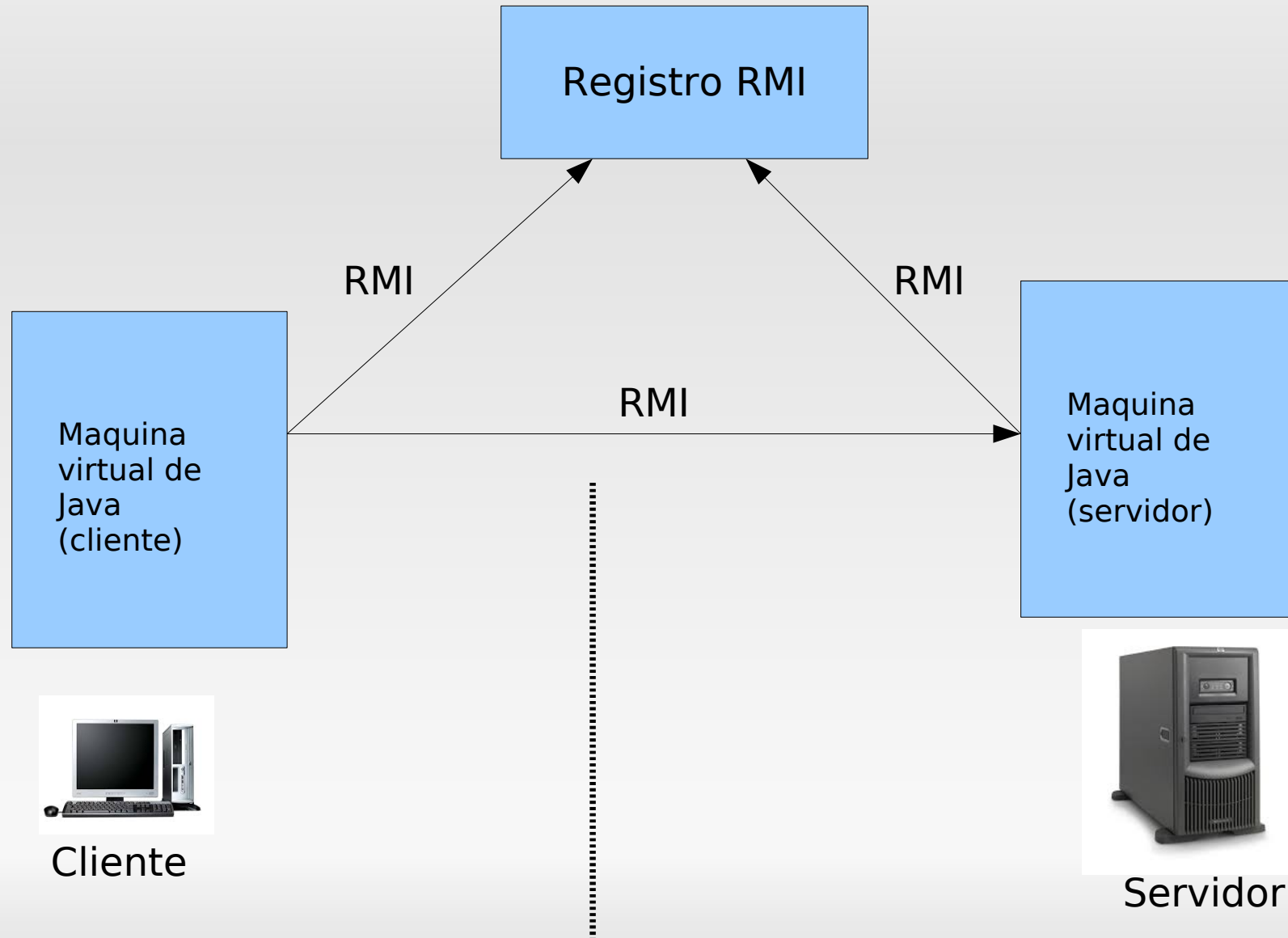
Cliente



Servidor

RMI – Remote Method Invocation

Registro RMI



RMI – Remote Method Invocation

Utilidad

- Ejecutar métodos en equipos remotos.
- Aplicaciones distribuidas.
- Aplicaciones especializadas.

RMI – Remote Method Invocation

Características

- Basado en objetos
- No es independiente del lenguaje, tanto el servidor como el cliente tienen que ser escritos en Java.
- La localización de los recursos se realiza mediante URL
- URL: Uniform Resource Locator: cadena de caracteres única que identifica un recurso de la red. Puede incorporar protocolo, dirección servidor, puerto, ruta (nombre archivo), ... Ejemplo:
 - <http://www.ehu.es:80/p202-home/eu/>

RMI – Componentes 1

- **Stub:** hace de intermediario encargándose del envío de la petición y de la recepción de la respuesta. Se encarga de:
 - Iniciar una conexión con la MV que contiene al objeto remoto.
 - Empaquetar (marshals) y transmitir los parámetros de la invocación a la MV remota.
 - Esperar los resultados de la invocación.
 - Desempaquetar (unmarshals) y devolver el valor de retorno o la excepción.
 - Devolver el valor a quien lo llamó.
- **Skeleton:** En la JVM remota, cada objeto remoto tiene su correspondiente esqueleto. Es responsable de enviar la llamada a la implementación del objeto remoto. Los pasos que realiza son:
 - Lee (deserializa) los parámetros del método remoto
 - Invoca el método en la implementación del objeto remoto
 - Escribe (serializa) y transmite el resultado
- Se generan a través del compilador **rmic**: con el sufijo `_Stub`.
- En las últimas versiones de Java los Stubs se generan dinámicamente en tiempo de ejecución y se hace innecesario el uso de los skeletons-

RMI – Componentes 2

- Registro RMI:
 - Las clases remotas a utilizar se tienen que registrar en el registro RMI para que el cliente las pueda localizar.
 - El comando `rmiregistry` permite iniciar el registro.
 - El registro de las clases se puede realizar utilizando la clase `Naming`.
 - Los métodos más importantes son:
 - `bind`: para registrar una clase remota.
 - `rebind`: para volver a registrar una clase remota.
 - `unbind`: para eliminar del registro la clase remota.
 - `list`: obtiene una lista con los objetos rmi registrados
 - `lookup`: obtiene desde el cliente un objeto remoto para poder utilizarlo desde una interfaz remota.

RMI — Pasos para crear una aplicación

- **Lado cliente:**
 - 1) Crear una interfaz con los métodos remotos heredada de la clase “**Remote**” que relance las excepciones RemoteException.

```
import java.rmi.*;

public interface InterfaceRemota extends Remote {
    public int sumar (int a, int b) throws RemoteException;
}
```

RMI — Pasos para crear una aplicación

- **Lado cliente:**

2) En el main de la aplicación cliente crear un objeto de la interfaz llamando al método “lookup” de la clase “Naming” con la dirección URL de la clase remota en el servidor:

```
public class Cliente {  
  
    /** Crea nueva instancia de Cliente */  
    public Cliente()  
    {  
        try  
        {  
            InterfaceRemota objetoRemoto =  
                (InterfaceRemota)Naming.lookup ("//localhost/ObjetoRemoto");  
  
            // Se realiza la suma remota.  
            System.out.print ("2 + 3 = ");  
            System.out.println (objetoRemoto.sumar(2,3));  
        }  
        catch (Exception e)  
        {  
            System.out.println("Excepción: " + e.getMessage());  
        }  
    }  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        new Cliente();  
    }  
}
```

RMI — Pasos para crear una aplicación

- **Lado servidor:**

1) Crear la interfaz remota igual que en el lado del cliente y la clase que implemente la interfaz remota y herede de `UnicastRemoteObject` .

```
import java.io.Serializable;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ObjetoRemoto extends UnicastRemoteObject
    implements InterfaceRemota
{
    public ObjetoRemoto () throws RemoteException
    {
        super();
    }

    public int sumar(int a, int b)
    {
        System.out.println ("Sumando " + a + " + " + b +"...");
        return a+b;
    }
}
```


RMI — Pasos para crear una aplicación

- Lado servidor:

- 2) En el main .

```
import java.rmi.*;

public class Servidor
{
    /** Crea nueva instancia de Servidor rmi */
    public Servidor() {
        try
        {
            // Se publica el objeto remoto
            InterfaceRemota objetoRemoto = new ObjetoRemoto();
            Naming.rebind ("//localhost/ObjetoRemoto", objetoRemoto);
        }
        catch (Exception e)
        {
            System.out.println("Excepción: " + e.getMessage());
        }
    }

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

RMI — Pasos para crear una aplicación

- **Crear el Stub (lado cliente) y el el Skeleton (lado server) :**
Compilar el objeto remoto con el compilador **rmic** para obtener el fichero `ObjetoRemoto_Stub.class`
 - * A partir de la versión J2SE 1.2 de Java no son necesarios los Skeletons y a partir de la versión J2SE 5.0 no son necesarios los stub.
 - * Ojo al nombre de los paquetes: el paquete de las clases remotas debe tener el mismo nombre en el cliente y en el servidor.

RMI — Pasos para crear una aplicación

- **Ejecutar las aplicaciones:**

1) Iniciar el registro desde la consola en la carpeta donde están las clases compiladas .class (o utilizar la variable de entorno CLASSPATH) con:

`rmiregistry puerto &`

`ó`

`rmiregistry`

Para trabajar con el puerto estandar (1099)

2) Ejecutar el server

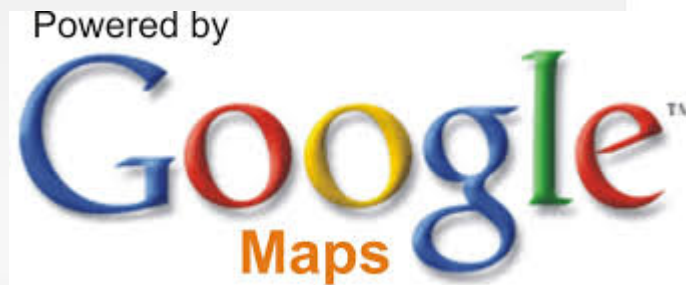
3) Ejecutar el cliente



Java web services

Web services — ¿Para qué?

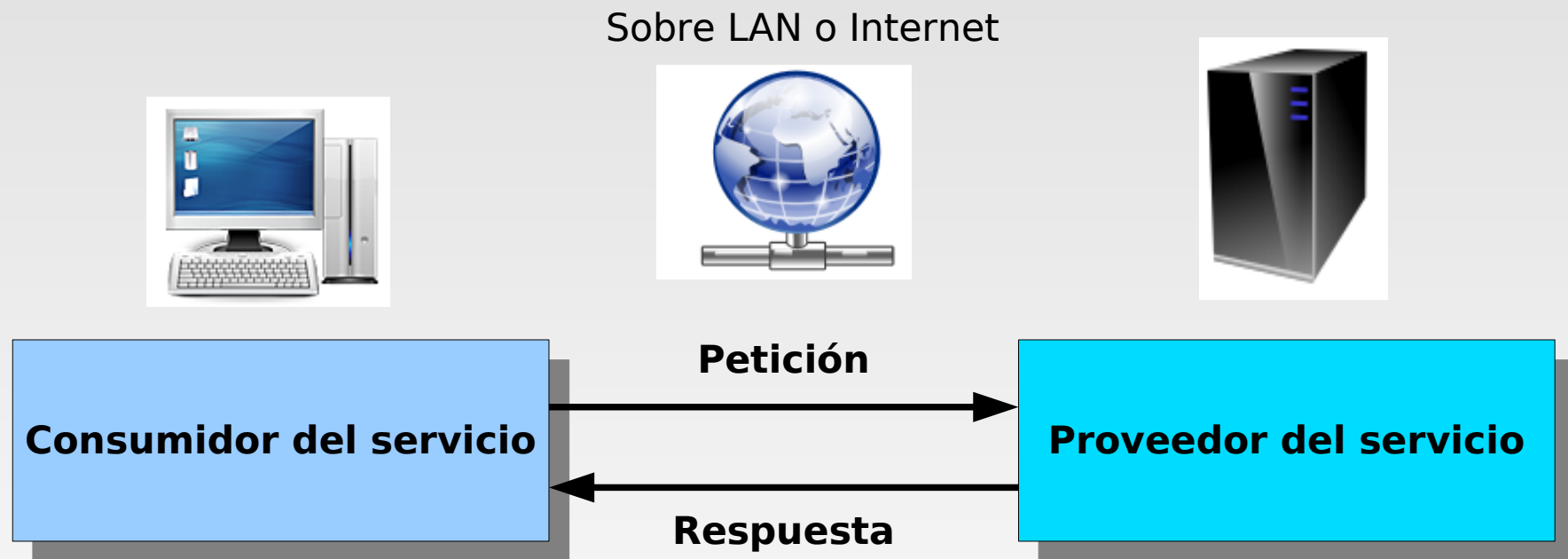
- Para traducir un texto: [Google Translate](#)
- Para gestionar una colección de fotos en la red: [Flickr](#)
- Obtener un mapa: [Google Maps](#)
- Realizar una búsqueda en la red: [Google Search](#)
- Consultar el tiempo: [Global weather](#)
- ...



Web services — Concepto 1

- Un servicio web o *Web service* es un conjunto de **protocolos y estándares** que sirven para intercambiar datos entre aplicaciones y solicitar servicios remotos a servidores.
- Distintas aplicaciones de software desarrolladas en **lenguajes de programación diferentes**, y **ejecutadas sobre cualquier plataforma**, pueden utilizar los servicios web para intercambiar datos en redes de ordenadores como Internet.
- La interoperabilidad se consigue mediante la adopción de estándares abiertos.
- Las organizaciones OASIS y **W3C** son los comités responsables de la arquitectura y reglamentación de los servicios Web.

Web services – Concepto 2



Ventajas

- Son **independientes** de la plataforma y del software (por ejemplo, un servidor Unix programado en Java puede ser explotado desde un cliente escrito en .NET de Microsoft).
- Al estar basados en **texto**, es fácil de entender su contenido y su funcionamiento.
- Permiten el **software distribuido** en diferentes compañías y lugares geográficos. Se crean servidores especializados en diferentes tareas: mapas, tiempo, fotos, finanzas, ...
- Normalmente se utiliza sobre **TCP en el puerto 80** reservado para los navegadores de Internet y por tanto no se encuentra filtrado por los firewalls.

Web services — Inconvenientes

- Su grado de desarrollo es inferior al de otros estándares abiertos de computación distribuida como CORBA (Common Object Request Broker Architecture).
- Inferior rendimiento comparando con otros modelos de computación distribuida, tales como RMI (Remote Method Invocation), CORBA o DCOM (Distributed Component Object Model). Sobre todo el uso del formato texto reduce ese rendimiento. El objetivo de XML (eXtensible Markup Language) no es ni la concisión ni la eficacia de procesamiento.
- Al apoyarse en HTTP y muchas veces sobre el puerto 80, pueden esquivar medidas de seguridad basadas en firewall, aunque ya existen firewalls especializados en XML.

Web services — Programación

- Los programadores de servicios web disponen de varios modelos de programación. Los modelos de programación se dividen en dos categorías:
 - Basados en **REST**(REpresentational State Transfer) es una nueva forma de crear y comunicarse con web services. En REST, los recursos tienen URIs (Uniform Resource Identifier) y son manipulados a través de operaciones de cabecera HTTP.
 - Basados en **SOAP/WSDL**: Es el método tradicional de programación de web services. Los servicios son publicados a través de ficheros WSDL (un tipo de XML) que tienen URLs. El intercambio de mensajes se realiza en SOAP (otro tipo de documento XML).

Web services – REST vs SOAP 1

REST

- Encaja con la arquitectura Internet
- Apropiada para clientes sencillos (JavaScript)
- No requiere Proxy para conversiones
- Basado en acceso a recursos (GET,PUT, ...)

SOAP

- Cifrado a nivel de mensaje
- Intermediarios (routing, seguridad)
- Mensajería confiable (reliably delivered)
- Políticas de control
- Basado en acciones (métodos)

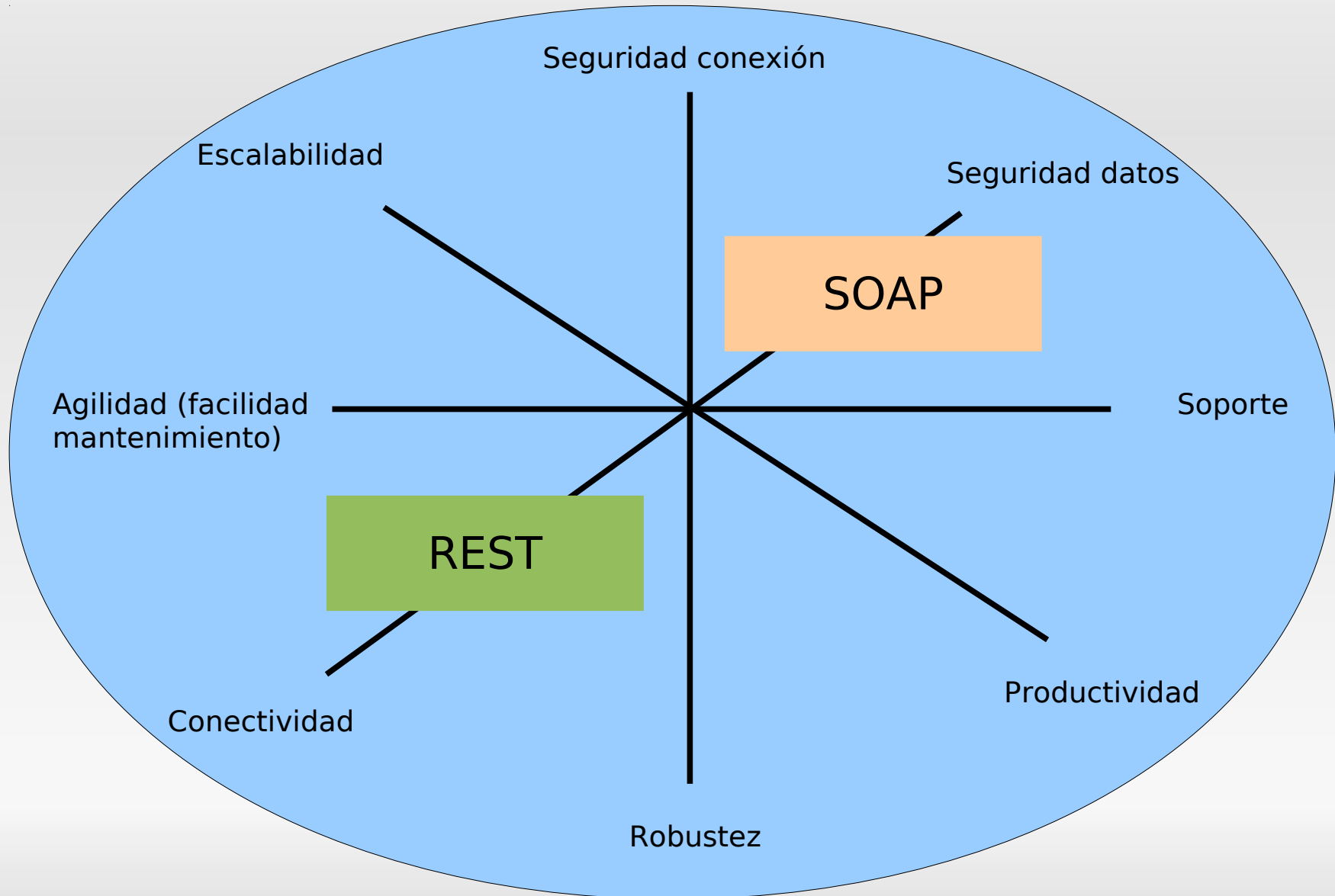
- Operaciones remotas
- Acceso a recursos remotos
- Interoperable

Aplicación Internet

Aplicación

Aplicación empresarial

Web services – REST vs SOAP



Web services — Basados en REST 1

- Son colecciones de recursos web identificados por URIs. Cada documento y cada proceso es modelado como un recurso web con una **URI única**. Las acciones que se pueden realizar con ese servicio se recogen en un fichero cabecera HTTP. El intercambio de mensajes se realiza en formato XML, JSON, HTML, etc. En muchos casos el cliente puede ser un navegador web.
- El protocolo de REST es HTTP. Se dispone sólo de 4 métodos: GET, PUT, POST, and DELETE. Las peticiones se pueden dejar marcadas y las respuestas se pueden almacenar en caché. Un administrador de red puede saber fácilmente que ofrece un servicio web con tan sólo mirar las cabeceras HTTP.
- REST es una tecnología válida en aquellos casos en los que requiere una seguridad no superior a la ofrecida por HTTP y en los que el protocolo HTTP es el apropiado. Así y todo REST puede ofrecer una funcionalidad sofisticada. Por ejemplo Flickr, Google Maps, Google Translate y Amazon proveen servicios REST: Ejemplo [Google Translate](#).

Web services — Basados en REST 2

- Métodos o “verbos” RESTful:
 - GET: Buscar y obtener información
 - PUT: Crear nueva información
 - POST: Editar información, modificar
 - DELETE: Borrar información

Web services – Características de REST

- **Cliente-servidor:** Se distingue el cliente y el servidor con funciones diferenciadas y repartidas. Por ejemplo el cliente no se preocupa de almacenar los datos, eso es tarea del servidor. De esa forma aumenta la portabilidad del código de los clientes. Los servidores no se preocupan del interfaz ni del estado del usuario, de esa forma el servidor puede ser más simple y escalable. El cliente y el servidor pueden ser desarrollados de forma independiente mientras no se altere el interfaz entre ellos.
- **Sin estado:** El servidor se despreocupa del estado del cliente. Cada petición por parte del cliente incluye toda la información necesaria para servirla y el estado de la sesión es controlado por el cliente.
- **Cacheable:** Como en el World Wide Web, los clientes pueden guardar en el caché las respuestas. Las respuestas se pueden definir como cacheables o no para evitar el uso de datos falsos en algunos casos. El uso del caché reduce las interacciones cliente-servidor y puede mejorar el rendimiento.
- **Sistema multinivel:** El cliente puede decidir si se conecta directamente al servidor final o a servidores intermediarios a lo largo del camino. Los servidores intermediarios proporcionan escalabilidad, equilibran la carga a través de cachés compartidos. Esto también puede reforzar las políticas de seguridad.
- **Código bajo demanda** (opcional) : Los servidores pueden temporalmente mejorar la funcionalidad de un cliente transfiriéndole código ejecutable, que se ejecuta en el lado cliente. Son ejemplos de esto los applets de Java y los scripts como JavaScript.
- **Interfaz unificada:** El interfaz unificado entre el cliente y el servidor simplifica la arquitectura y permite que parte (cliente y servidor) se desarrolle de forma independiente. El interfaz incluye: la identificación de recursos bajo demanda, capacidad para manipular los recursos del servidor desde el cliente, mensajes auto-descriptivos y cambios de estado en el cliente provocados por las respuestas ofrecidas por el servidor.

Web services – Ejemplo interacción REST



**Consumidor
del servicio**

Sobre LAN o Internet



**Proveedor
del servicio**

**Petición HTTP:
verbo + params**

GET
`https://www.googleapis.com/language/translate/v2?
key=KEY_ACCESO_GOOGLE&source=en&target=de&q=Hello
%20world`

**Respuesta:
en diferentes formatos (JSON, XML,
HTML, binario, etc...)**

200 OK

Código de estado HTTP

```
{  
  "data": {  
    "translations": [  
      {  
        "translatedText": "Hallo Welt"  
      }  
    ]  
  }  
}
```

Resultado en formato JSON

Web services – Ejemplo de petición HTTP /JSON

```
Firefox
https://maps.g...i&sensor=false
https://maps.googleapis.com/maps/api/directions/json?origin=Bilbao&destination=Donosti&sensor=false
Google

{
  "routes" : [
    {
      "bounds" : {
        "northeast" : {
          "lat" : 43.30343000000001,
          "lng" : -1.998010
        },
        "southwest" : {
          "lat" : 43.16728000000001,
          "lng" : -2.924110
        }
      },
      "copyrights" : "Mapako datuak: ©2013 Google, basado en BCN IGN España",
      "legs" : [
        {
          "distance" : {
            "text" : "95,2 km",
            "value" : 95152
          },
          "duration" : {
            "text" : "1 ordu, 3 minutu",
            "value" : 3782
          },
          "end_address" : "Donostia-San Sebastián, Gipuzkoa, Espainia",
          "end_location" : {
            "lat" : 43.302370,
            "lng" : -1.998010
          },
          "start_address" : "Bilbao, Bizkaia, Espainia",
          "start_location" : {
            "lat" : 43.25671000000001,
            "lng" : -2.924110
          },
          "steps" : [
            {
              "distance" : {
                "text" : "21 m",
                "value" : 21
              },
              "duration" : {
                "text" : "1 minutu",
                "value" : 6
              },
              "end_location" : {
                "lat" : 43.256540,
                "lng" : -2.924220
              },
              "html_instructions" : "Jarraitu \u003cb\u003ehego-mendebalde\u003c/b\u003e aldera \u003cb\u003eBelostikale Kalea\u003c/b\u003e kalean \u003cb\u003eTenderia Kalea\u003c/b\u003e norabidean",
              "polyline" : {
                "points" : "mq_gGtbzP@T"
              },
              "start_location" : {
                "lat" : 43.25671000000001,
                "lng" : -2.924110
              },
              "travel_mode" : "DRIVING"
            },
            {
              "distance" : {
```

Web services – Ejemplo de petición HTTP /XML

```
Firefox
https://maps.g...i&sensor=false
https://maps.googleapis.com/maps/api/directions/xml?origin=Bilbao&destination=Donosti&sensor=false
XML fitxategiak ez dirudi estilo-informaziorik duenik lotuta. Dokumentuaren zuhaitza behekaldean ikus daiteke

- <DirectionsResponse>
  <status>OK</status>
  - <route>
    <summary>AP-8</summary>
    - <leg>
      - <step>
        <travel_mode>DRIVING</travel_mode>
        - <start_location>
          <lat>43.2567100</lat>
          <lng>-2.9241100</lng>
        </start_location>
        - <end_location>
          <lat>43.2565400</lat>
          <lng>-2.9242200</lng>
        </end_location>
        - <polyline>
          <points>mq_gGtbzP`@T</points>
        </polyline>
        - <duration>
          <value>6</value>
          <text>1 minutu</text>
        </duration>
        - <html_instructions>
          Jarraitu <b>hego-mendebalde</b> aldera <b>Belostikale Kalea</b> kalean <b>Tenderia Kalea</b> norabidean
        </html_instructions>
        - <distance>
          <value>21</value>
          <text>21 m</text>
        </distance>
      </step>
    </leg>
  </route>
  - <step>
    <travel_mode>DRIVING</travel_mode>
    - <start_location>
      <lat>43.2565400</lat>
      <lng>-2.9242200</lng>
    </start_location>
    - <end_location>
      <lat>43.2563900</lat>
      <lng>-2.9238100</lng>
    </end_location>
    - <polyline>
```

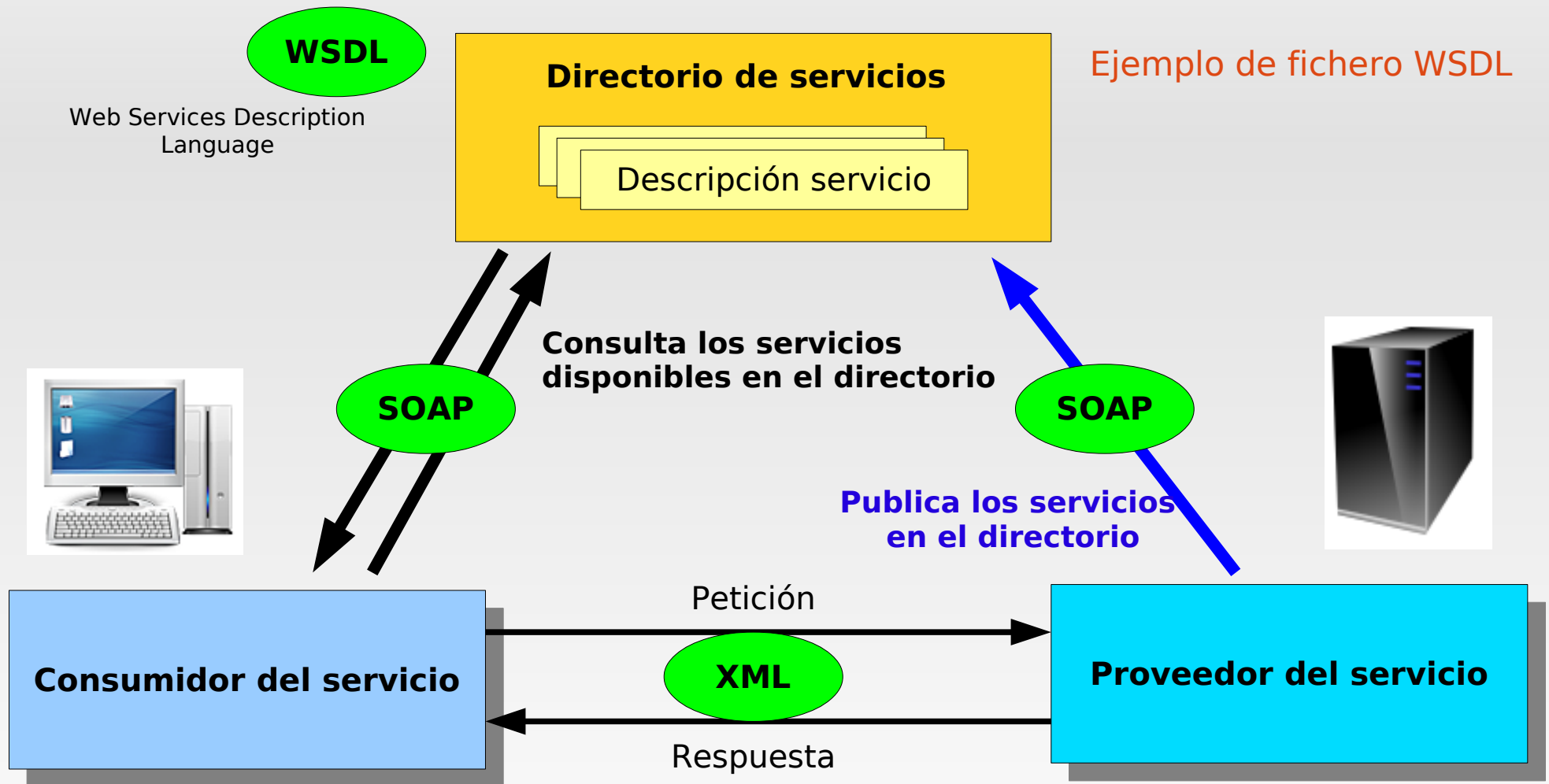
Tutorial "Getting Started with RESTful Web Services"

- Ejemplo de creación de un servidor RESTful sobre una base de datos JavaDB y servidor GlassFish.
- Test del servicio utilizando NetBeans.
- Ejemplo de una aplicación cliente RESTful sobre GoogleMaps.

Web services — Basados en SOAP/WSDL

- Los servicios son publicados en un fichero WSDL en la red. Los interesados en utilizar ese servicio crean un cliente basado en ese fichero.
- Los mensajes se intercambian en formato SOAP (Simple Object Access Protocol).
- El rango de operaciones que se puede realizar en SOAP es mayor que en REST, especialmente en seguridad.
- Los servicios web basados en SOAP son indicados para aplicaciones de peso que utilizan operaciones complicadas y que requieren seguridad sofisticada y fiable. Además se pueden utilizar protocolos que no sean HTTP. Las aplicaciones relacionadas con transacciones comerciales están basadas en SOAP.

Web services — Modelo basado en SOAP/WSDL



SOAP: Simple Object Access Protocol: Protocolo para que dos objetos en diferentes procesos puedan intercambiar datos XML

XML: eXtensible Markup Language: Metalenguaje extensible de etiquetas: manera de definir lenguajes para diferentes necesidades. Simplificación del SGML.

Tutorial de NetBeans “Getting Started with JAX-WS Web Services”

- Ejemplo de creación de un servicio web SOAP-WSDL que suma dos números sobre un servidor GlassFish.
- Test del servicio utilizando NetBeans.
- Ejemplo de una aplicación Java cliente para consumir ese servicio.
- Ejemplo de una aplicación cliente SOAP-WSDL sobre GlobalWeather.

Web services — Librerías de Java

- Clase **URL**: Uniform Resource Locator. Puntero a un recurso de la WWW.
- Clase **URI**: Uniform Resource Identifier.
- Clase **HttpURLConnection**: permite realizar peticiones a un servidor HTTP.
- Paquete **javax.xml.parsers**: permite manejar documentos XML
- **JAX-WS**: API de Java para servicios web basados en XML.

Web services — HTTP / SOAP / XML

XML

- Es un lenguaje.
- Basado en texto.
- Pensado para almacenar, transportar e intercambiar datos.
- No pensado para visualizar datos (frente a HTML).
- Diferentes utilidades: bases datos,
- Estructura: Header (es XML + versión) , solo 1 root element, varios elementos child, tags de cierre <etiqueta > </etiqueta >.

HTTP

- Protocolo a nivel de aplicación
- Es el encargado de enviar

SOAP

- Protocolo para intercambiar información estructurada.
- El formato de mensaje es XML.
- El envío y la negociación de los mensajes se realiza normalmente sobre HTTP, HTTPS o SMTP.
- Se encarga de que lo que se envía pueda ser un “Web service”

Web services — Ejemplo mensaje SOAP

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice xmlns:m="http://www.example.org/stock">
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

Web services – Glosario terminos 1

Término	Def. acrónimo	Qué es?	Descripción
WEB	-	Varias cosas	Red, malla, telaraña
WSDL	Web Services Description Language	Lenguaje	Describe un servicio web: procedimientos, parámetros de entrada-salida, mecanismos de encapsulado y transporte (p.e: SOAP sobre HTTP)
SGML	Standard Generalized Markup Language	Metalenguaje	Sistema para la organización y etiquetado de documentos
XML	eXtensible Markup Language	Metalenguaje	Metalenguaje extensible de etiquetas: manera de definir lenguajes para diferentes necesidades. Simplificación del SGML.
HTML	HyperText Markup Language	Lenguaje	Lenguaje de marcado muy usado para la creación de páginas web. Permite incluir información, dar formato, incluir scripts de código, Definido en términos de SGML.
XHTML	eXtensible HyperText Markup Language	Lenguaje	HTML expresado como XML válido
SOA	Service-Oriented Architecture	Arquitectura	Arquitectura de software que define la utilización de servicios para dar soporte a los requisitos del negocio.
SOAP	Simple Object Access Protocol	Protocolo	Protocolo para que dos objetos en diferentes procesos puedan intercambiar datos XML

Web services – Glosario terminos 2

Término	Def. acrónimo	Qué es?	Descripción
UDDI	Universal Description, Discovery, and Integration	Estándar	Estándar básico de los servicios Web cuyo objetivo es ser accedido por los mensajes SOAP y dar paso a documentos WSDL, en los que se describen los requisitos del protocolo y los formatos del mensaje solicitado para interactuar con los servicios Web del catálogo de registros.
ASP.NET	Active Server Pages	Framework	Plataforma de programación de Microsoft para la creación de servicios web XML, aplicaciones web y sitios web dinámicos.
W3C	World Wide Web Consortium	Consorcio	Consorcio internacional que emite recomendaciones para la WWW
REST	REpresentational State Transfer	Modelo programación	Es una nueva forma de crear y comunicarse con web services. En REST, los recursos tienen URIs y son manipulados a través de operaciones de cabecera HTTP

Web services – Glosario terminos 3

Término	Def. acrónimo	Qué es?	Descripción
JAXB	Java Architecture for XML Binding	API de Java	Permite mapear clases de Java a representaciones XML. Ayuda para programar con documentos XML.
JSON	JavaScript Object Notation	Formato de intercambio de datos	Subconjunto de la notación literal de objetos de JavaScript que no requiere el uso de XML.
JavaScript	-	Lenguaje de programación	Interpretado y orientado a objetos, se utiliza para insertar código en páginas web.
AJAX	Asynchronous JavaScript And XML	Técnica de desarrollo web	Permite crear aplicaciones interactivas basado en estándares abiertos como JavaScript y Document Object Model (DOM)