Lenguaje java

Ing° Pedro Beltrán Canessa
pbeltranc@uladech.pe

Licencia de Uso

Acerca de este documento

Ing° Pedro Beltrán Canessa
pbeltranc@uladech.pe

Este documento ha sido realizado íntegramente con software libre.

Índice

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

Introducción a la OOP

Conceptos

- Herencia: posibilidad de que una clase adquiera atributos y/o métodos de otra ya existente, pudiendo además añadirse otros miembros, y redefinirse métodos heredados. A la clase madre se le llama "superclase" y a la hija "subclase".
- Polimorfismo: métodos con igual nombre implementan distintas funcionalidades según las clases sobre las que se apliquen. Concepto relacionado: sobrecarga
- Encapsulamiento: ocultación de código y datos de objetos

Características generales del lenguaje

- Es mucho más que un lenguaje
- 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
- Nacido en la década de 1990. Antepasado: OAK

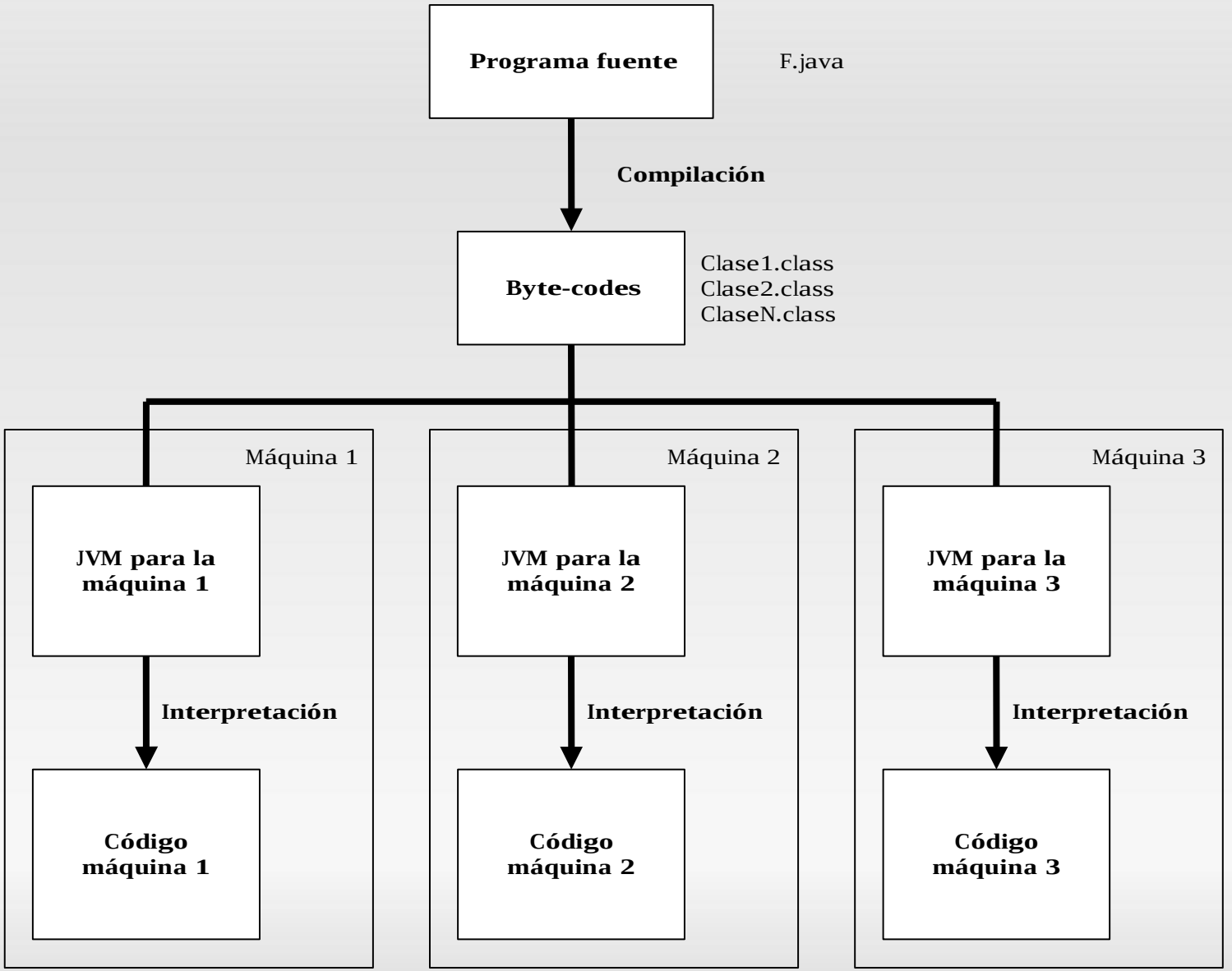
Introducción a la tecnología java

Características generales del lenguaje

- Orientado a objetos
- Lenguaje compilado (bytecodes) e interpretado (JVM).
  - JVM: Es el intérprete de bytecodes a código binario nativo de cada plataforma.
  - 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 remota, no en local

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

Comparación con C

- Algunas diferencias con C:
  - No tiene preprocesado ni ficheros de cabecera
  - No hay estructuras
  - Gestión de memoria automática (“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)

Aplicaciones

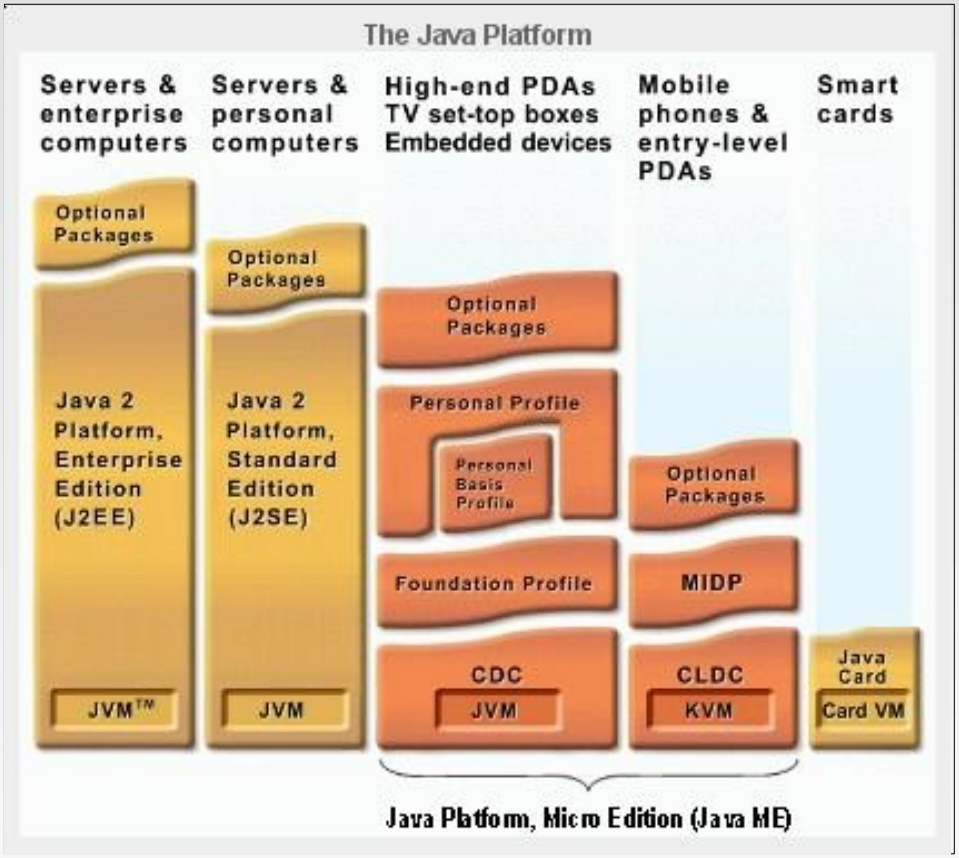
- Programas “standalone” convencionales. Ej.
  - Software para teléfonos móviles
  - Software para PC (Windows o GNU/Linux), ej. azureus
  - Software para servidor, ej. páginas web dinámicas generadas por un servidor que ejecute servlets o jsp
- Applets: programas ejecutados en un navegador web que tenga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

Interfaces gráficos de usuario

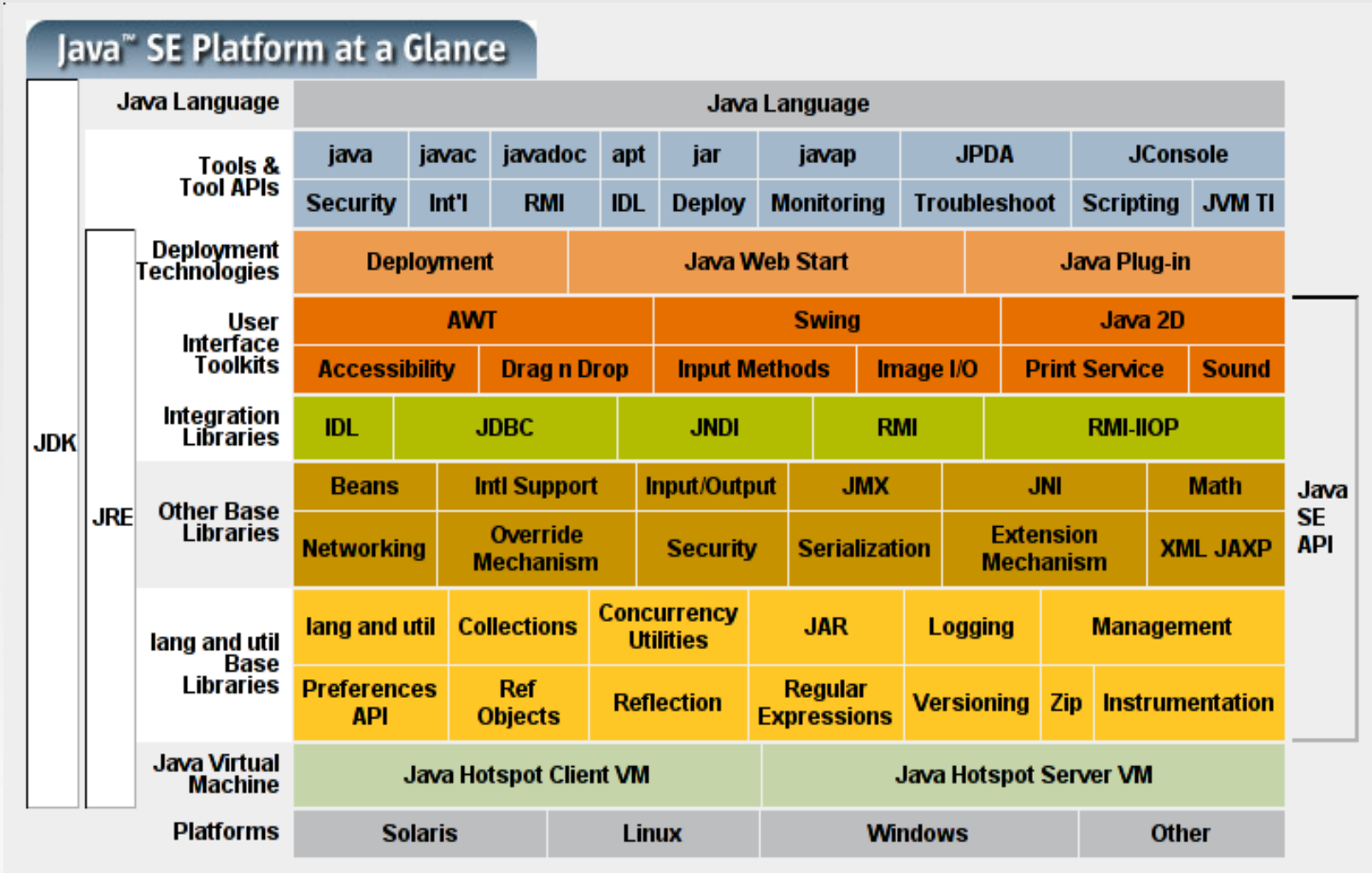
- Principales:
  - AWT (de Sun) es el más antiguo
  - Swing (de Sun)
  - SWT (de IBM Eclipse). Apariencia de la plataforma sobre la que corre.
- Bindings para entornos gráficos específicos, ej. KDE/qt, GNOME/gtk,...

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



Tecnologías Java de Sun - Familia SE



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

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

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,...

Procedimiento básico de desarrollo

- Edición del fuente, ej:

```
emacs NombreClasePublica.java
```

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

```
javac NombreClasePublica.java
```

- Ejecución: la variable de sistema CLASSPATH debe estar al valor adecuado (ver capítulo de paquetes).

```
java NombreClasePublica
```

En el caso de que sea una clase no perteneciente al paquete por defecto:

```
java nombrePaquete.NombreClasePublica
```

Sintaxis básica

Características generales

- Lenguaje "case-sensitive"
- Tipo de líneas de un fichero fuente: además de las de C, nuevo comentario:
  - Una o más líneas entre `/**` y `*/` para generar documentación

Sintaxis básica

Variables y constantes

- Constan de:
  - Identificador o etiqueta.
  - Valor. Valor inicial.
  - 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.

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 ni modulares)

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`)

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

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

Variables y constantes – Tipos referencia

- Un tipo referencia indica que en la variable se guardará un objeto o instancia de una clase.
- Ocupan el nº 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 de tipos distintos; ver anexo de herencia y conversiones entre tipos referencia) ej:

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

Sintaxis básica

Literales

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

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 ,

Operadores - Tabla de precedencia

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

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)

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
void mostrarInt(int i)	void mostrar(int i)
void mostrarLong(long l)	void mostrar(long l)
void mostrarFloat(float f)	void mostrar(float f)

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
```

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

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

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

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.

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)

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

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.

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

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(); }
}
```

- Si se define algún otro constructor para una clase, Java no crea el constructor por defecto y es necesario definirlo si se quiere permitir inicializar objetos llamando al constructor por defecto.

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);
...
```

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.

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 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

Herencia

Introducción

- Definición:
  - posibilidad de que una clase adquiera 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.

Herencia

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.

Herencia

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.
- 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 !!!
    }
}
```

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, nº 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, nº 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")

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 `ArrayIndexOutOfBoundException`

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.
- 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. ejs:

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

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

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

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 `ArrayIndexOutOfBoundException`)
  - 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, is Empty, size, setSize

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
```

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.Object

    ```
    String toString()
    ```

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

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

- 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:
  - Compilación:
    - Sentencias de importación: en el fuente donde se usen, importar las clases e interfaces=indicar a qué paquete o paquetes pertenecen.
    - Sentencias de uso
  - Runtime: Localización de los ficheros .class en el sistema operativo, indicando dónde se encuentran los directorios de los ficheros .class de los paquete a emplear, mediante la variable de entorno CLASSPATH. 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

- 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

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

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

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.

Clases para Excepciones

- En java una excepción se modela como una clase que desciende 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

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 irrecuperables y se finaliza el programa.

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, ejs: 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 excepciones mediante código

- Se lanzan excepciones mediante código haciendo:

```
throw <objetoDeClaseExcepcion>
```

Ej: `throw new Exception();`

- Un método declarado de la forma:

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

significa que puede lanzar esa lista de excepciones separadas por comas, implícitas o explícitas.
(Más adelante se verá cuándo es obligatorio)

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

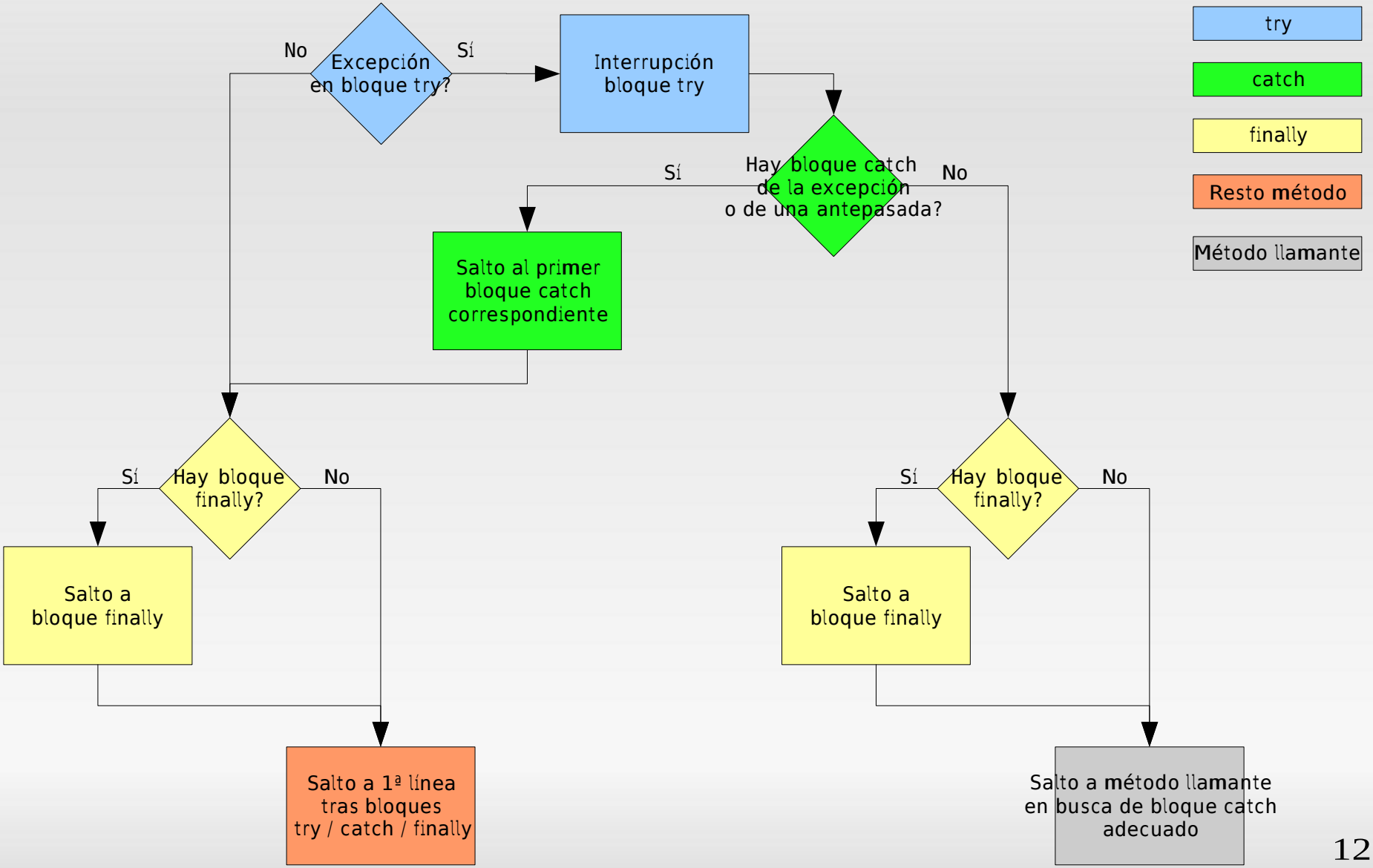
Secuencia de una excepción – Lanzamiento

- Durante la ejecución de un método se pueden producir o "lanzar" (throw) excepciones de dos formas:
  - La JVM detecta una situación especial, ej:
    - Software: una división por cero, una llamada a un método en el que se produce una excepción no tratada en el mismo.
    - Hardware: ej. un error al cargar una parte del programa.
  - Un método lanza una excepción por código (throw)
- En ambos casos se emplea un objeto de la clase correspondiente a la excepción (en el primer caso lo crea la JVM).

Secuencia de una excepción – Tratamiento

- A continuación, 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.

Secuencia de una excepción – Diagrama de Flujo



Excepciones explícitas - Lanzamiento

- En un método se considera que se pueden producir excepciones explícitas de dos formas:
  - Lanzamiento con `throw <objetoExcepcionExplicita>`
  - Llamada a método declarado con `throws <ExcepcionesExplicitas>`

Excepciones explícitas – Tratamiento

- Se produce un error de compilación si en un bloque try se pueden lanzar excepciones explícitas (con throw o llamando a otro método declarado con throws EExpl), y el 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 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 (ambos casos con throw o llamando a otro método declarado con throws EExpl).
- Lo anterior no aplica para la clase Exception.

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 explícitas – Declaración método con throws

- De forma obligatoria, para propagar excepciones explícitas a métodos llamantes:
  Si en el método se pueden producir excepciones explícitas (con throw o al llamar a otro método declarado con throws EExpl) y no se tratan dentro (catch para las mismas o sus ascendientes).
- De forma opcional, como previsión para subclases:
  Si se quiere 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).