



Tema 4. Programación de aplicaciones en red con Java

Introducción a las redes de ordenadores

Boni García
Curso 2017/2018

Índice de contenidos

1. Introducción a Java
2. Elementos básicos
3. Clases y objetos
4. Elementos avanzados
5. Introducción a Eclipse
6. Gestión de entrada/salida
7. Colecciones
8. Programación con sockets
9. Hilos en Java
10. Para ampliar

Índice de contenidos

1. **Introducción a Java**
 - ¿Qué es Java?
 - Gestión de memoria en Java
 - Historia de Java
 - Ecosistema Java
2. Elementos básicos
3. Clases y objetos
4. Elementos avanzados
5. Introducción a Eclipse
6. Gestión de entrada/salida
7. Colecciones
8. Programación con sockets
9. Hilos en Java
10. Para ampliar

1. Introducción a Java

¿Qué es Java?



- Java es un lenguaje de programación de alto nivel:
 - **Independiente de plataforma** (*cross-platform*). Se dice que Java es un lenguaje compilado e interpretado. El código fuente Java es compilado a un lenguaje llamado *bytecode* que posteriormente es interpretado en una máquina virtual de Java (*Java Virtual Machine*, JVM). Este principio se conoce como WORA (*Write Once, Run Anywhere*)
 - **Sencillo**. La memoria es gestionada de forma automática (no existen los punteros). La liberación de recursos no utilizados la realiza un componente de la JVM llamado recolector de basura (*garbage collector*)
 - **Multihilo**. Un programa Java puede dividir su ejecución en diferentes tareas (hilos) que se ejecutan simultáneamente

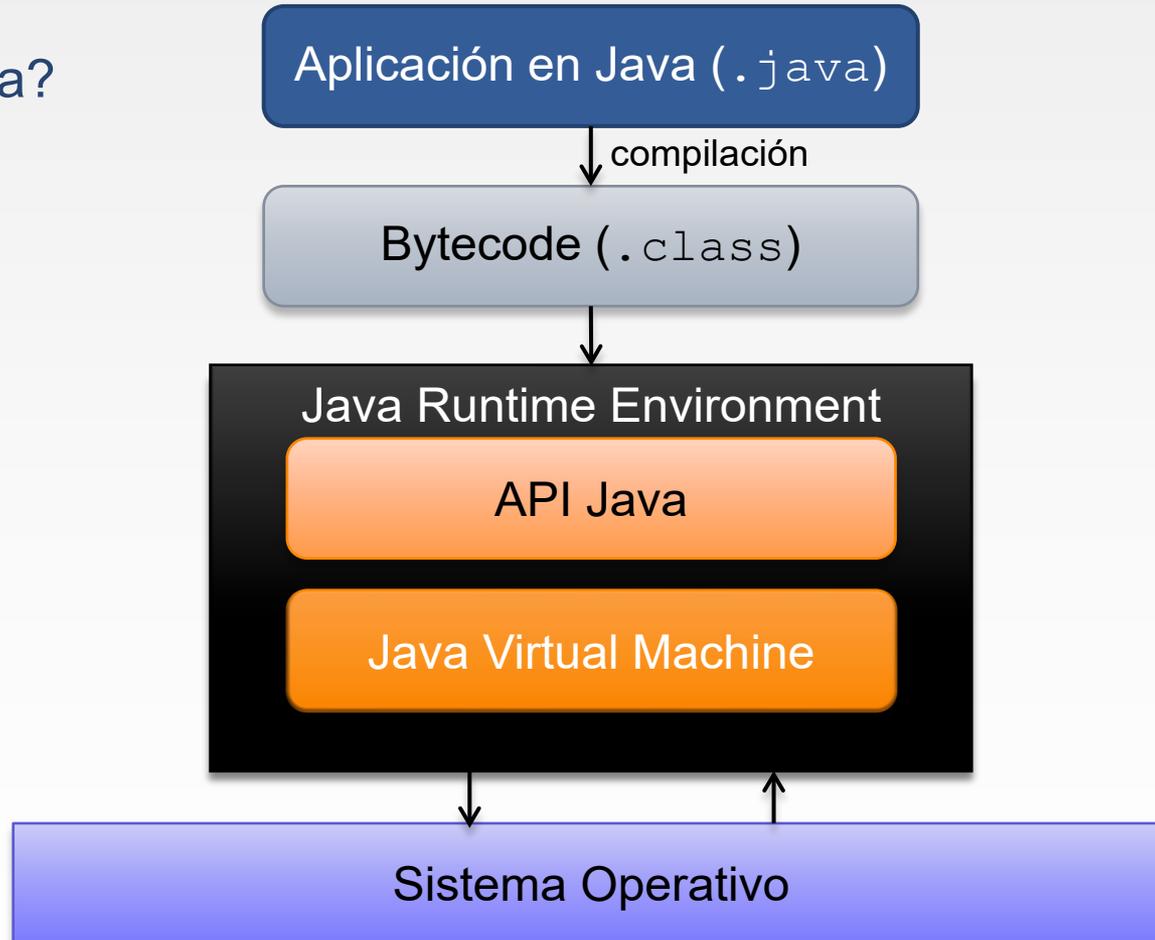
1. Introducción a Java

¿Qué es Java?

- En cuanto a **paradigma**, podemos decir que Java es un lenguaje principalmente **imperativo** (basado en instrucciones que se usan para componer algoritmos) y además **orientado a objeto** (las aplicaciones Java se diseñan e implementan usando clases, que se instancian en memoria como objetos)
 - Desde su versión 8, Java además incorpora características más propias del paradigma **funcional** (es un tipo de programación declarativa basada en el uso de funciones matemáticas). Por ejemplo el uso de funciones *lambda* o las colecciones de tipo *stream*
- En cuanto a **sistema de tipos**, se dice que Java implementa tipado **estático** (la comprobación de tipos se hace en tiempo de compilación)

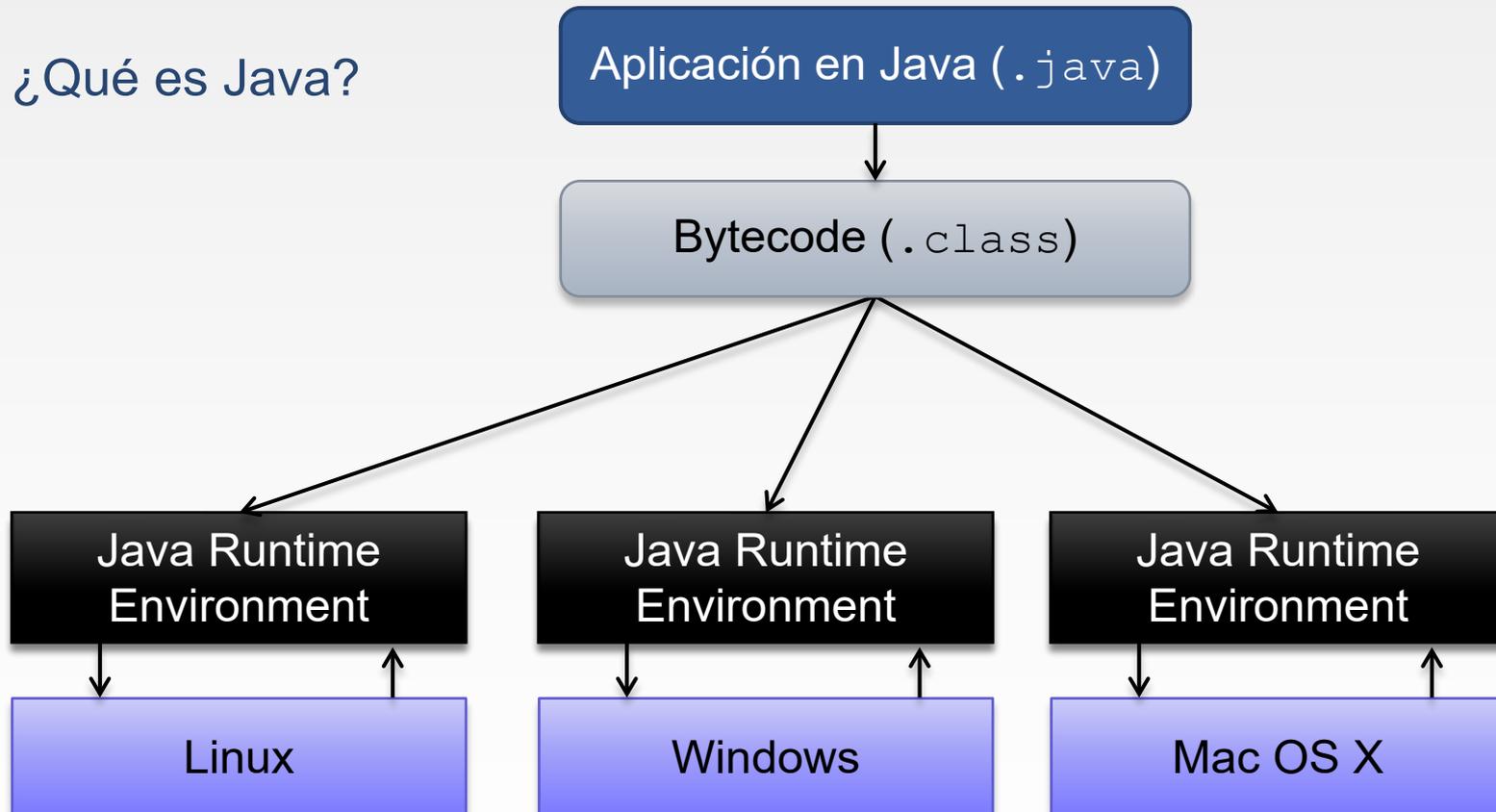
1. Introducción a Java

¿Qué es Java?



1. Introducción a Java

¿Qué es Java?



...

1. Introducción a Java

¿Qué es Java?

- Una API (Interfaz de programación de aplicaciones, *Application Programming Interface*) es el conjunto de funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por un componente software
- La API de Java proporciona un conjunto de clases para efectuar toda clase de tareas dentro de un programa. Por ejemplo:
 - Operaciones de entrada/salida
 - Colecciones (listas, mapas, ...)
 - Comunicación entre procesos remotos (sockets)
 - Gestión de hilos
 - ...

1. Introducción a Java

¿Qué es Java?

- El entorno de ejecución Java (Java Runtime Environment, **JRE**) incluye todo lo necesario para ejecutar aplicaciones Java
 - Descargable desde <http://www.java.com>
 - Incluye la máquina virtual JVM (*Java Virtual Machine*) y la API de Java
 - Incluye un cargador de clases (*class loader*) que carga dinámicamente las clases Java en la JVM
 - No contiene el compilador
- El entorno de desarrollo Java (Java Development Kit, **JDK**) incluye al JRE, el compilador (`javac`) y otras herramientas para desarrollar aplicaciones

1. Introducción a Java

¿Qué es Java?

- Existen varias ediciones Java, diseñadas para ámbitos de ejecución en particular. Las principales son:
 1. Java Standard Edition (Java SE)
 - Versión usada en ordenadores personales
 2. Java Enterprise Edition (Java EE)
 - Define un conjunto de librerías para el desarrollo de aplicaciones web
 - Necesita Java SE para ejecutarse
 3. Java Micro Edition (Java ME)
 - Entorno para desarrollar aplicaciones en dispositivos con recursos limitados como dispositivos móviles, software empotrado (por ejemplo electrodomésticos, etc)
 - Existe una distribución todavía más ligera: Java Card (para tarjetas inteligentes)

1. Introducción a Java

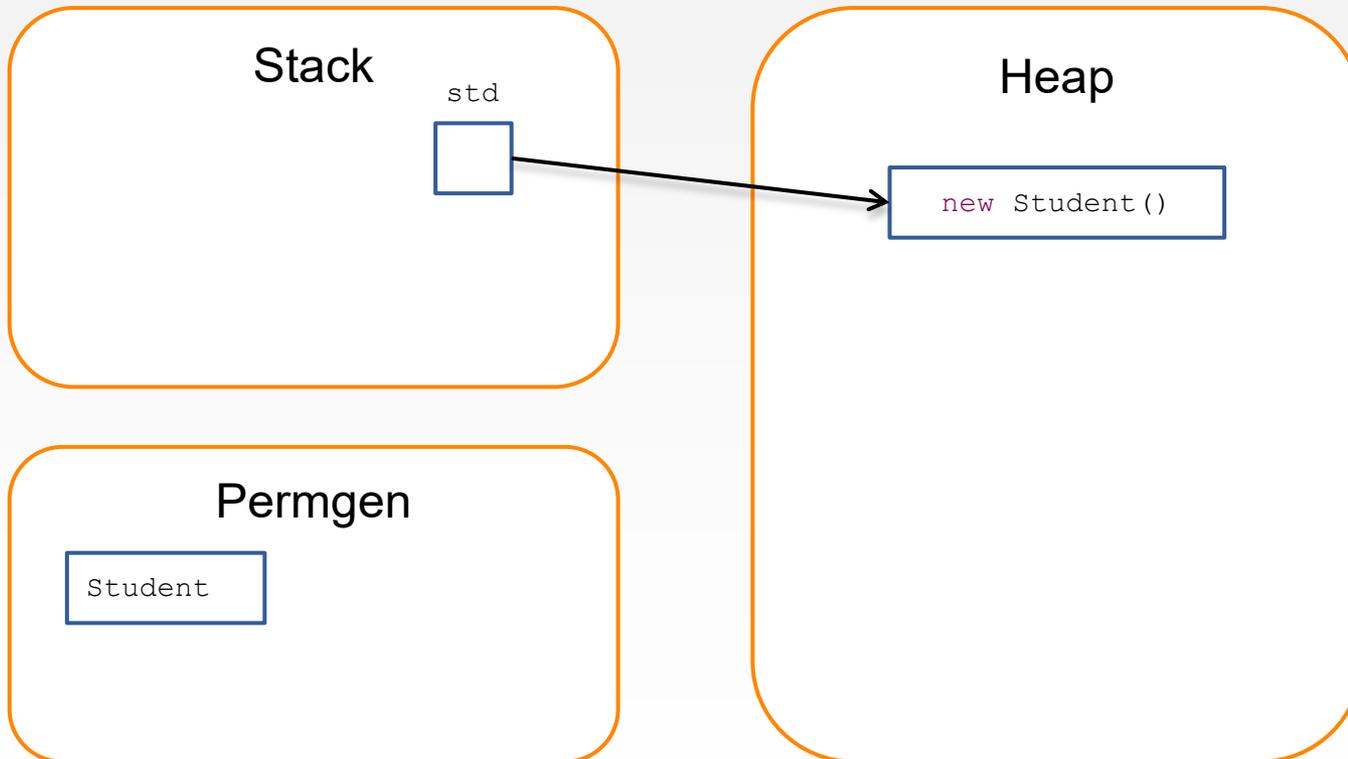
Gestión de memoria en Java

- El **heap** es el espacio de memoria en tiempo de ejecución que se usa para almacenar los objetos Java
 - Se crea para cada proceso Java y es gestionado por el recolector de basura
 - Cuando se reserva espacio de almacenamiento en la pila para un objeto, este sigue vivo mientras exista una referencia a él
- El **stack** es el espacio de memoria donde se almacenan las referencias a objetos
- El **permgen** es el espacio de memoria donde se almacena la información de las clases

1. Introducción a Java

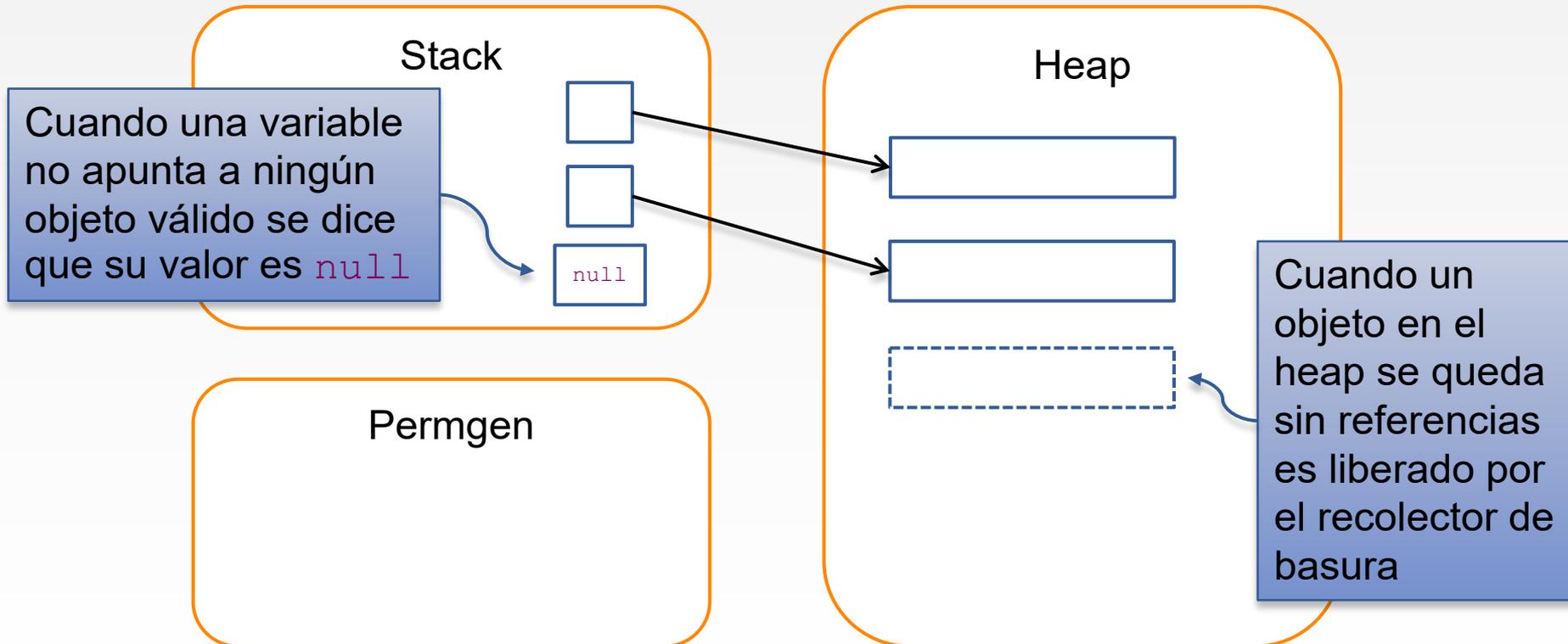
Gestión de memoria en Java

```
Student std = new Student();
```



1. Introducción a Java

Gestión de memoria en Java



1. Introducción a Java

Historia de Java

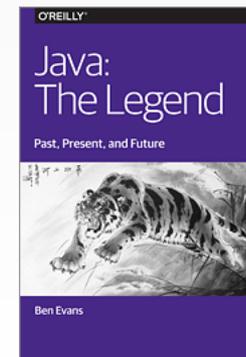
- 1990: Varios ingenieros de Sun Microsystems (James Gosling, Mike Sheridan, Patrick Naughton) no estaban satisfechos con C++ (principalmente con la gestión manual de la memoria) y comienzan a desarrollar un nuevo lenguaje de programación
 - Aplicado a la electrónica de consumo
 - Lenguaje fácil de aprender y usar
 - Basado en C++
 - Multiplataforma: principio WORA (*Write Once, Run Anywhere*)
 - Escriben el compilador “Oak”

1. Introducción a Java

Historia de Java

- 1994: Se orienta el proyecto hacia la web, cambiando el nombre de “Oak” a Java
- 1996: Sun Microsystems forma la empresa JavaSoft. Aparece JDK 1.0
- 2006: Java se libera como *open source*
- 2007: Se crea el OpenJDK
- 2009: Oracle compra Sun Microsystems

Más información sobre la historia de Java en el libro gratuito “*Java: The Legend*” (Ben Evans, O’Reilly, septiembre 2015)



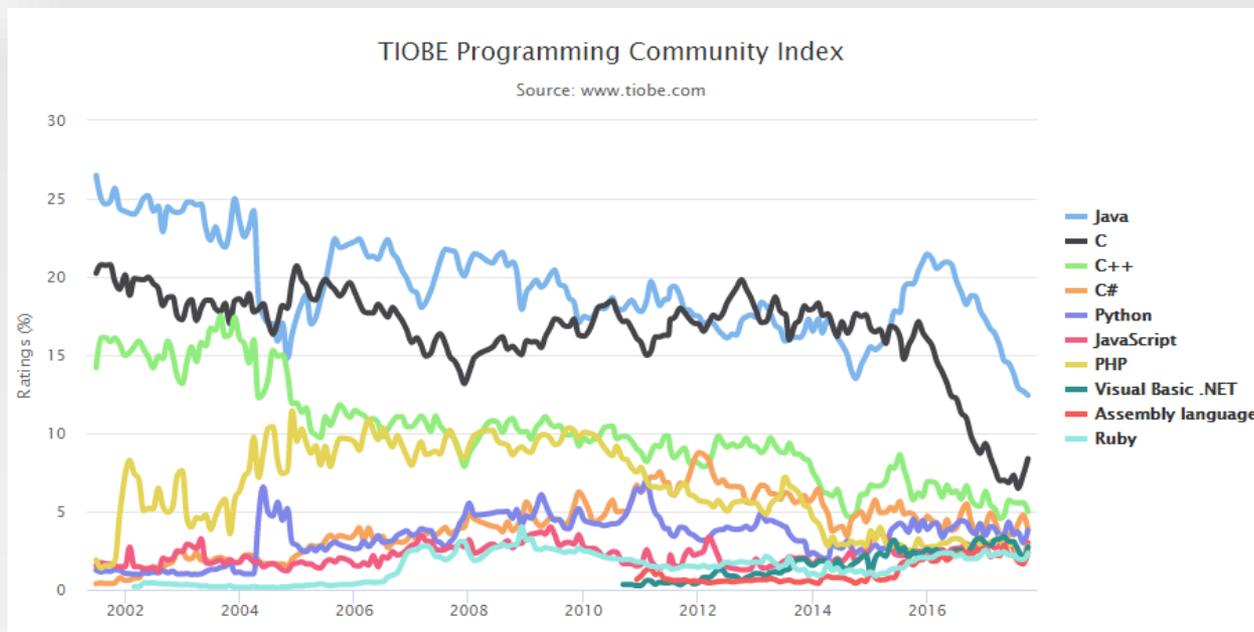
1. Introducción a Java

Historia de Java

Versión	Fecha	Características
JDK 1.0	1996	Primera versión estable de Java
JDK 1.1	1997	Clases internas, JDBC, RMI
J2SE 1.2	1998	Swing, colecciones
J2SE 1.3	2000	Soporte de depuración
J2SE 1.4	2002	API de entrada salida, expresiones regulares
J2SE 5.0	2004	Anotaciones, genéricos, enumerados, JIT
Java SE 6	2006	Mejora soporte GUI, servicios web
Java SE 7	2011	Nueva API de entrada salida, protocolos de red
Java SE 8	2014	Expresiones lambdas
Java SE 9	2017	Modularidad

1. Introducción a Java

Historia de Java



<http://www.tiobe.com/tiobe-index/>

El **índice TIOBE** es un indicador que mide la popularidad de los lenguajes de programación

Se calcula usando el número de búsquedas en la Web de los diferentes lenguajes

1. Introducción a Java

Ecosistema Java

- Un IDE (*Integrated Development Environment*) es un programa que diseñado para ayudar en el proceso de desarrollo de software
- En Java, los más utilizados son:
 - **Eclipse** (www.eclipse.org)
 - Software libre
 - Desarrollado por la Fundación Eclipse (apoyada por IBM)
 - **Netbeans** (www.netbeans.org)
 - Software libre
 - Desarrollado por Oracle
 - **IntelliJ IDEA** (www.jetbrains.com/idea)
 - Software libre con versión profesional de pago



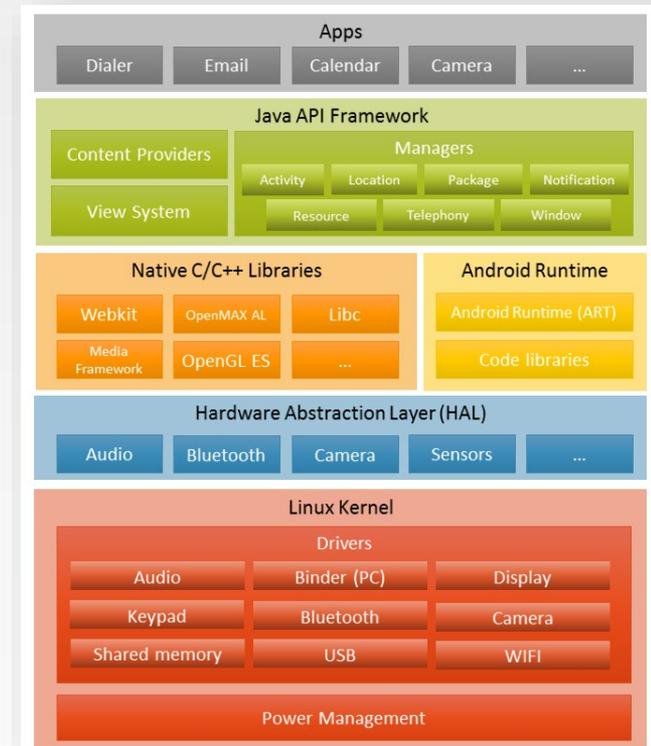
1. Introducción a Java

Ecosistema Java

- **Android** es un sistema operativa para dispositivos móviles basado en una versión modificada de Linux
- Las aplicaciones Android (*apps*) se implementan con lenguaje Java
- La máquina virtual no es la *Java Virtual Machine* (JVM), es otra denominada *Android Runtime* (ART) desde Android 4.4 (anteriormente Dalvik)
- El entorno de desarrollo para aplicaciones se llama Android SDK (*Software Developer Kit*)



<http://www.android.com/>



1. Introducción a Java

Ecosistema Java

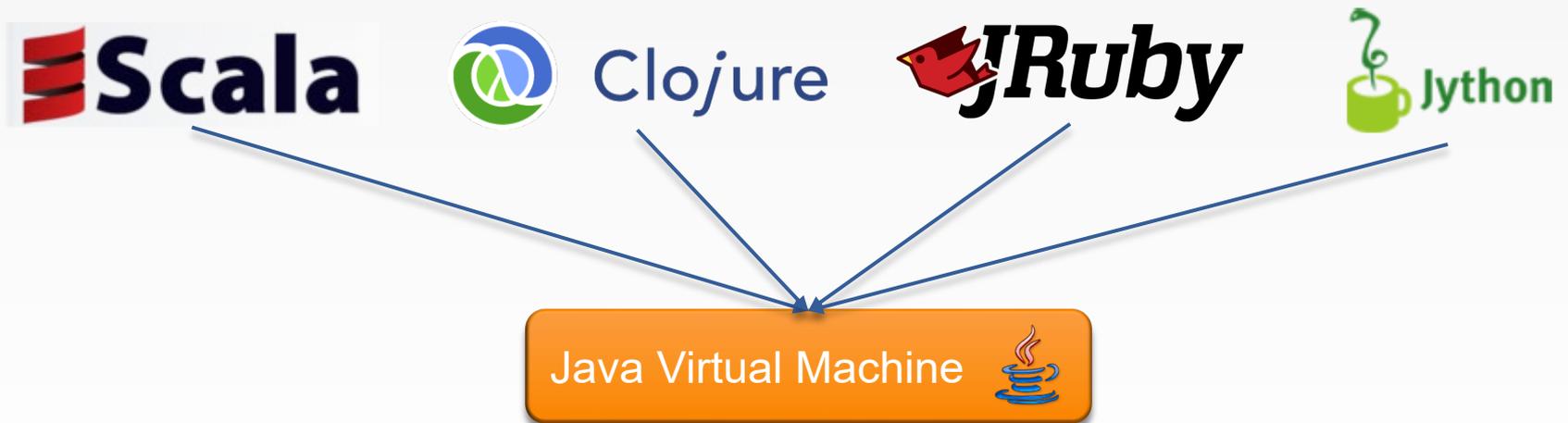
- **Spring** es un *framework* de alto nivel muy utilizado para el desarrollo de aplicaciones Java
- La fundación **Apache** (*Apache Software Foundation, ASF*):
 - Es una organización sin ánimo de lucro dedicada al software
 - Han desarrollado el servidor web más usado en la actualidad (Apache)
 - En Java también desarrollan muchas utilidades importantes (Maven, Tomcat, Apache Commons, ...)



1. Introducción a Java

Ecosistema Java

- Otros lenguajes que se ejecutan en la máquina virtual Java: **Scala**, **Clojure**, **JRuby**, **Jython**



Índice de contenidos

1. Introducción a Java
2. Elementos básicos
 - Tipos de datos primitivos
 - Cadenas en Java
 - Comentarios
 - Operadores
 - Sentencias de control
 - Arrays
 - Compilador e intérprete Java
 - Primer programa en Java: HelloWorld
3. Clases y objetos
4. Elementos avanzados
5. Introducción a Eclipse
6. Gestión de entrada/salida
7. Colecciones
8. Programación con sockets
9. Hilos en Java
10. Para ampliar

2. Elementos básicos

Tipos de datos primitivos

Tipo	Contenido	Valor por defecto	Bits	Rango de valores
boolean	Valor lógico	false	1	true/false
char	Carácter Unicode	\u0000	16	\u0000 a \uFFFF
byte	Entero	0	8	-128 a 127
short	Entero	0	16	-32768 a 32767
int	Entero	0	32	-2147483648 a +2147483647
long	Entero	0L	64	-9223372036654775806 a 9223372036854775807
float	Decimal	0.0f	32	$-3.4028235 \cdot 10^{38}$ to $3.4028235 \cdot 10^{38}$ (6 dígitos de precisión)
double	Decimal	0.0o	64	$-1.7976931348623157 \cdot 10^{308}$ to $1.7976931348623157 \cdot 10^{308}$ (15 dígitos de precisión)

2. Elementos básicos

Cadenas en Java

- Se utiliza la clase `String`:

```
String string = "This is an example of string";
```

Las sentencias
en Java
acaban en ;

- Se comparan con `equals`:

```
String string1 = "Hello";  
String string2 = "Goodbye";  
boolean comparation = string1.equals(string2);
```

- Los objetos de tipo `String` en Java con **inmutables** (una vez que se crean no pueden cambiar de valor)

Los objetos inmutables son una forma de código simple y seguro (por ejemplo en situaciones de concurrencia)

2. Elementos básicos

Comentarios

```
// One-liner comment

/* Block of several lines of
   comments */

/**
 * Comments for JavaDoc.
 */
```

2. Elementos básicos

Operadores

Aritméticos

Operador	Descripción
+	Suma
-	Resta
*	Multiplica
/	Divide
%	Resto

Relacionales

Operador	Descripción
>	Mayor
<	Menor
>=	Mayor o igual que
<=	Menor o igual que
==	Igual
!=	Distinto

Condicionales

Operador	Descripción
&&	And
	Or

- Java también utiliza el operador + para concatenar cadenas de caracteres

2. Elementos básicos

Operadores

De manipulación de bits y lógica entre bits

Operador	Uso	Operación
>>	op1 >> op2	Desplaza los bits de op1 a la derecha op2 veces
<<	op1 << op2	Desplaza los bits de op1 a la izquierda op2 veces
&	op1 & op2	Operación AND
	op1 op2	Operación OR
^	op1 ^ op2	Operación XOR
~	~ op	Complemento

2. Elementos básicos

Operadores

De asignación directa

Operador	Uso	Operación
<code>+=</code> , <code>-=</code> , <code>/=</code> , <code>*=</code> , <code>%=</code>	<code>op1 += op2</code>	<code>op1 = op1 + op2</code>
<code><<=</code> , <code>>>=</code> , <code>^=</code>	<code>op1 <<= op2</code>	<code>op1 = op1 << op2</code>

Operador ternario (“Elvis”)

Operador Elvis	Uso
<code>? :</code>	<code>op = (boolean_condition) ? value1 : value;</code>

2. Elementos básicos

Sentencias de control

- Condicional `if-else`

```
if (booleanCondition) {  
    // code to be executed if the condition is true  
} else {  
    // code to be executed if the condition is false  
}
```

- Ejemplo

```
public class IfElseExample {  
    public static void main(String args[]) {  
        int money = 300;  
        if (money > 0) {  
            System.out.println("You have " + dinero + " euros");  
        } else {  
            System.out.println("You don't have any money");  
        }  
    }  
}
```

2. Elementos básicos

Sentencias de control

- Bucle `while`

```
while (booleanCondition) {  
    // It remains in the loop while the condition is true.  
    // Normally in the body of the loop the condition is modified  
    // to provide an exit to the loop (otherwise it is an infinite  
    // loop)  
}
```

- Ejemplo

```
public class WhileExample {  
    public static void main(String args[]) {  
        int i = 5;  
        while (i >= 0) {  
            System.out.println(i);  
            i--;  
        }  
    }  
}
```

2. Elementos básicos

Sentencias de control

- Bucle `do-while`

```
do {  
    // The same as while, but in this case the body of the loop  
    // is executed at least once  
} while (booleanCondition);
```

- Ejemplo

```
public class DoWhileExample {  
    public static void main(String args[]) {  
        int i = 0;  
        do {  
            if (i < 5) {  
                System.out.println(i);  
                i++;  
                continue;  
            }  
            break;  
        } while (true);  
    }  
}
```

2. Elementos básicos

Sentencias de control

- Bucle `for`

```
for (init; condition; step) {  
    // body  
}
```

- Ejemplo

```
public class ForExample {  
    public static void main(String args[]) {  
        for (int i = 0; i < 5; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

2. Elementos básicos

Sentencias de control

- Instrucción `switch`

```
switch (variable) {  
  case valor:  
    // ...  
    break;  
  case valor:  
    // ...  
    break;  
  default:  
    // ...  
}
```

La variable utilizada como opción del `switch` puede ser `byte`, `char`, `short`, `int` o `long`, y desde Java 7 también `String`.

Si no se incluye la clausula `break` al terminar el caso se continúa ejecutando el siguiente.

2. Elementos básicos

Sentencias de control

- Ejemplo `switch`

```
public class SwitchExample {
    public static void main(String args[]) {
        int qualification = 9;
        String result = "";
        switch (qualification) {
            case 0:
            case 1:
            case 2:
            case 3:
            case 4:
                result = "Suspenso";
                break;
            case 5:
            case 6:
                result = "Aprobado";
                break;
            case 7:
            case 8:
                result = "Notable";
                break;
            case 9:
            case 10:
                result = "Sobresaliente";
                break;
            default:
                result = "Error";
        }
        System.out.println("Has sacado un " + nota + " (" + resultado + ")");
    }
}
```

2. Elementos básicos

Arrays

- Son estructuras de datos que permiten almacenar una lista de un mismo tipo identificados por un índice

```
public class ArrayExample {
    public static void main(String args[]) {
        // Declaración
        String[] arrayStrings = new String[10];
        char arrayChars[] = new char[5];
        int[] arrayInts = { 3, 2, 1 };
        int[][] arrayMultiDim = new int[2][3];

        // Acceso a sus valores
        System.out.println(arrayInts[2]);

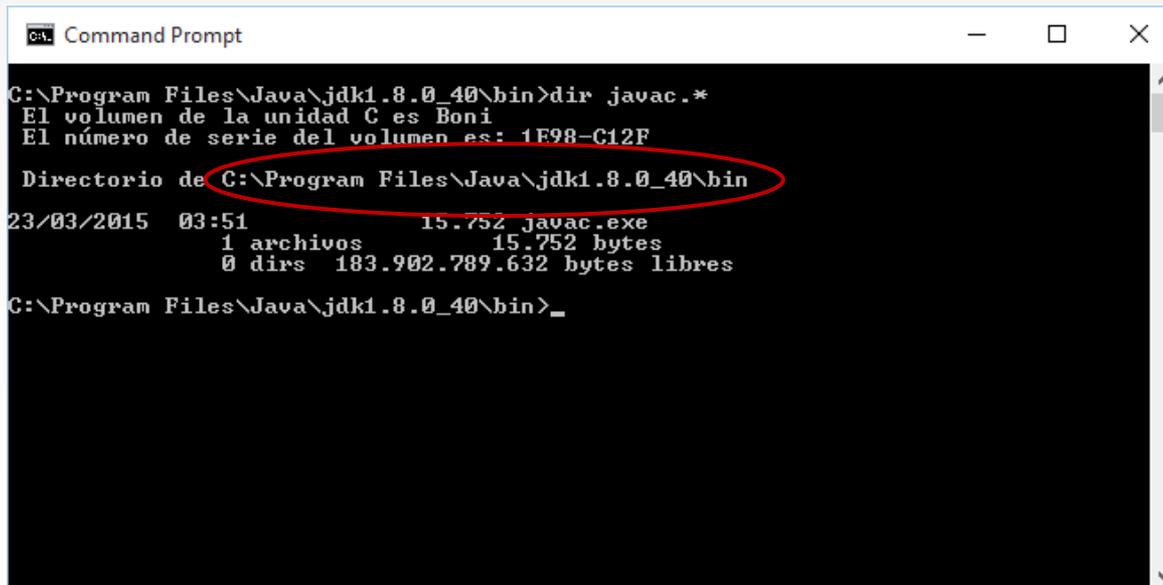
        // Modificación
        arrayStrings[0] = "Hello";
        System.out.println(arrayStrings[1]);
        arrayMultiDim[0][0] = 4;
        System.out.println(arrayMultiDim[0][0]);

        // Tamaño
        System.out.println(arrayChars.length);
    }
}
```

2. Elementos básicos

Compilador e interprete Java

- El **compilador** en Java SE es una aplicación llamada **javac**
- Podemos encontrar esta aplicación en el directorio `bin` de la instalación de la JDK



```
Command Prompt
C:\Program Files\Java\jdk1.8.0_40\bin>dir javac.*
El volumen de la unidad C es Boni
El número de serie del volumen es: 1F98-C12F

Directorio de C:\Program Files\Java\jdk1.8.0_40\bin
23/03/2015  03:51                15.752  javac.exe
             1 archivos                15.752 bytes
             0 dirs 183.902.789.632 bytes libres

C:\Program Files\Java\jdk1.8.0_40\bin>
```

2. Elementos básicos

Compilador e interprete Java

- La sintaxis de `javac` es la siguiente:

```
javac [options] [source files]
```

- Opciones más importantes:
 - `-classpath` o `-cp`: ruta a otras clases o paquetes Java ya compilados necesarios
 - `-d`: directorio destino de la compilación
 - `-version`: Escribe por pantalla la versión del compilador

2. Elementos básicos

Compilador e interprete Java

- El **interprete** en Java SE es una aplicación llamada **java**
- Podemos encontrar esta aplicación en el directorio `bin` de la instalación de la JDK o JRE

```
Command Prompt
C:\Program Files\Java\jdk1.8.0_40\bin>dir java.exe
El volumen de la unidad C es Boni
El número de serie del volumen es: 1E98-C12F
Directorio de C:\Program Files\Java\jdk1.8.0_40\bin
23/03/2015  03:51          206.728 java.exe
             1 archivos          206.728 bytes
             0 dirs 183.902.011.392 bytes libres
C:\Program Files\Java\jdk1.8.0_40\bin>_
```

```
Command Prompt
C:\Program Files\Java\jre1.8.0_40\bin>dir java.exe
El volumen de la unidad C es Boni
El número de serie del volumen es: 1E98-C12F
Directorio de C:\Program Files\Java\jre1.8.0_40\bin
23/03/2015  03:52          206.760 java.exe
             1 archivos          206.760 bytes
             0 dirs 183.901.155.328 bytes libres
C:\Program Files\Java\jre1.8.0_40\bin>_
```

2. Elementos básicos

Compilador e interprete Java

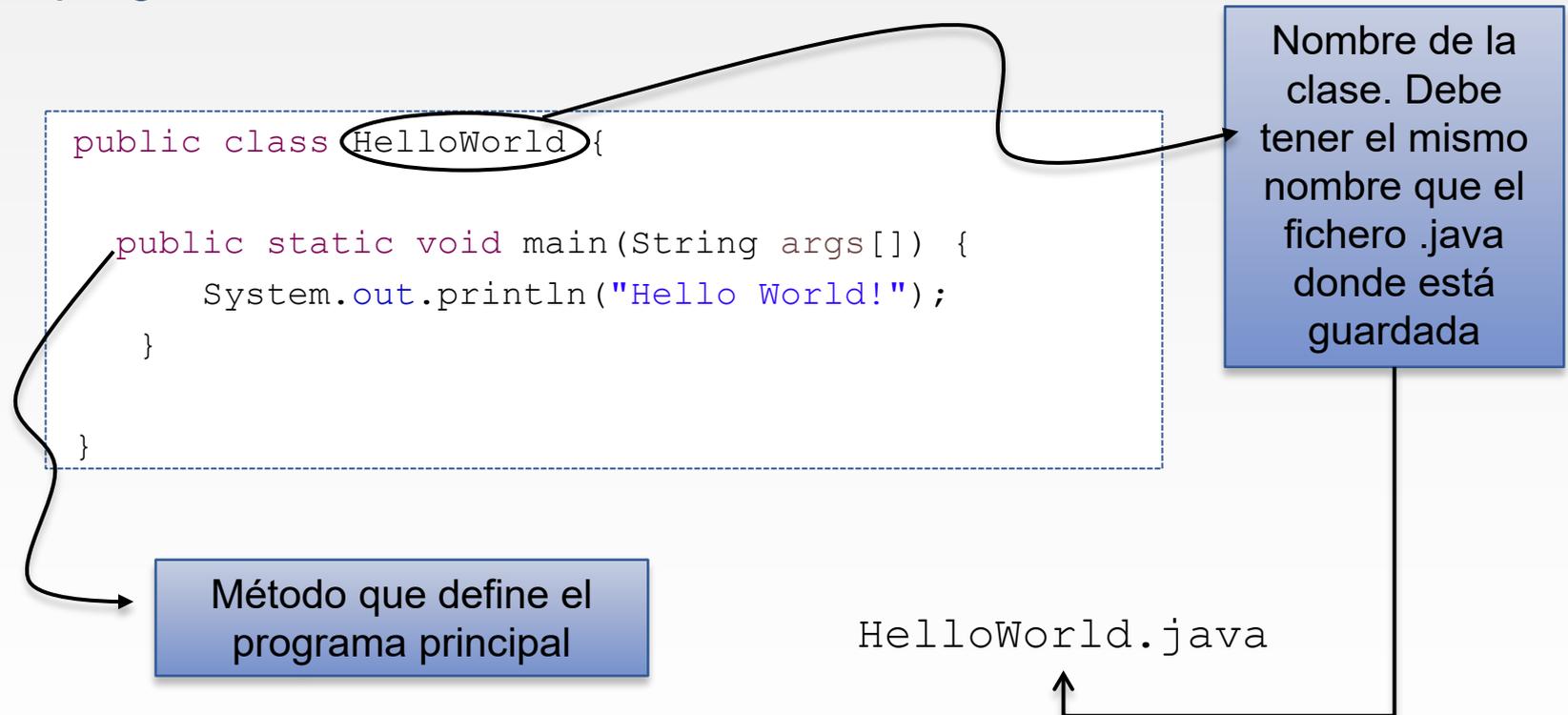
- La sintaxis de `java` es la siguiente:

```
java [options] class [arguments]
```

- Opciones más importantes:
 - `-classpath` o `-cp`: ruta a otras clases o paquetes Java ya compilados necesarias
 - `-jar <file.jar>`: Fichero JAR (*Java Archive*) en el cual encontramos la clase a ejecutar
 - `-version`: Escribe por pantalla la versión del intérprete
 - `-Dkey=value`: Propiedades de la máquina virtual. Pueden ser leídas desde las aplicaciones Java mediante el comando `System.getProperty("key")`
- Los argumentos se reciben como un array de cadenas en la clase que se está ejecutando (`public static void main(String args[])`)

2. Elementos básicos

Primer programa en Java: HelloWorld



Índice de contenidos

1. Introducción a Java
2. Elementos básicos
3. Clases y objetos
 - Programación orientada a objetos
 - Estructura de una clase Java
 - Paquetes
 - Definición de atributos, constructores y métodos
 - Modificadores de acceso
 - Métodos y atributos estáticos
 - Métodos, atributos, y variables finales
 - Creación de objetos
 - Reglas de estilo
 - Documentación
 - La API de Java
4. Elementos avanzados
5. Introducción a Eclipse
6. Gestión de entrada/salida
7. Colecciones
8. Programación con sockets
9. Hilos en Java
10. Para ampliar

3. Clases y objetos

Programación orientada a objetos

- La programación orientada a objetos (POO) es un paradigma de programación (estilo de programación) en el cual los programas se organizan como conjuntos de **objetos**
 - Una **clase** es el patrón que define a los objetos
 - Los **objetos** son instancias de las clases (son a menudo abstracciones a menudo provenientes del mundo real)
- Las clase están formadas por:
 - Atributos → estado del objeto
 - Métodos → comportamiento del objeto
- En un sistema orientado a objetos, los objetos interactúan entre sí, comunicándose para conseguir la funcionalidad esperada

3. Clases y objetos

Programación orientada a objetos

- Los principales principios de la POO son:
 - **Abstracción**
 - Separación del interfaz (especificación) de la implementación
 - **Encapsulación**
 - Reunir todos los elementos pertenecientes a una misma entidad
 - La base de la encapsulación es la clase
 - **Herencia**
 - Propiedad a través de la cual los objetos se relacionan dentro de una jerarquía
 - Una clase se crea a partir de otra (la extiende)
 - **Polimorfismo**
 - Capacidad de una entidad de referenciar distintos elementos
 - Concepto de polisemia del mundo real: un único nombre para muchos significados, según el contexto

3. Clases y objetos

Programación orientada a objetos

- Los tipos de polimorfismo son:
 - **Polimorfismo de asignación**
 - Variable que se declara como de un tipo pero que referencia en realidad un valor de un tipo distinto
 - **Sobrecarga** (polimorfismo ad-hoc, *overloading*)
 - Un solo nombre de método y muchas implementaciones distintas
 - **Sobreescritura** (polimorfismo de inclusión, *overriding*)
 - Tipo especial de sobrecarga que ocurre dentro de relaciones de herencia
 - **Genericidad**
 - Forma de crear clases o métodos de propósito general y especializarlas para situaciones específicas
 - Consiste en parametrizar clases con el objetivo de hacer código más robusto

3. Clases y objetos

Estructura de una clase Java

- Definición de paquete propio
- Definición de paquetes importados
- Documentación (JavaDoc)
- Definición clase (nombre del fichero .java donde se guarda la clase igual al nombre de la clase)
- Definición de los atributos
- Definición del constructor/constructores
- Definición de los métodos

```
package com.utad.idcd.redes;

import java.util.Scanner;

/**
 * Cartesian plane point (X,Y).
 *
 * @autor U-Tad
 * @version 1.0.0
 */
public class Point {
    // Attribute(s)
    private double x, y;

    // Constructor(s)
    public Point(double px, double py) {
        x = px;
        y = py;
    }
    public Point() {
        this(0.0, 0.0);
    }

    // Method(s)
    public double getX() {
        return x;
    }
    public void setX(double x) {
        this.x = x;
    }
}
```

```
public double getY() {
    return y;
}
public void setY(double y) {
    this.y = y;
}

public double distance(Point target) {
    return Math.sqrt((this.getX() - target.getX())
* (this.getX() - target.getX())
+ (this.getY() - target.getY())
* (this.getY() - target.getY()));
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Insert value for X: ");
    double x = sc.nextDouble();
    System.out.print("Insert value for Y: ");
    double y = sc.nextDouble();
    Point origin = new Point();
    Point target = new Point(x, y);
    System.out.print("The distance to origin is "
+ origin.distance(target));
    sc.close();
}
}
```

3. Clases y objetos

Paquetes

- Un paquete en Java es un contenedor de clases que permite agrupar las distintas partes de un programa con funcionalidad común
- Los paquetes se definen de forma jerárquica, separando niveles con puntos
- Esta estructura jerárquica se ve reflejada en una estructura de directorios para almacenar las clases
- Se usa la palabra reservada `package` para especificar a qué paquete pertenece una clase en la primera sentencia del programa
- Si no se especifica ninguno se asume el paquete anónimo
- Para usar una clase de otro paquete se usa la declaración `import`

```
import java.util.Scanner;  
import java.io.*;
```

- Por defecto, está importado el paquete `java.lang`

3. Clases y objetos

Definición de atributos, constructores y métodos

- Definición de atributo(s)

```
<modificador(s)> [type] [attributeName];
```

- Definición de constructor(es)

```
public ClassName (type1 paramName1, ...) {  
  
}
```

- Definición de método(s)

```
<modificador(s)> [tipo-devuelto] methodName (type1 paramName1, ...) {  
  
}
```

- Si el método no devuelve nada se usa la palabra reservada `void`
- En otro caso, el método deberá terminar su ejecución devolviendo una variable del tipo declarado con la palabra reservada `return`

Los modificadores de acceso discriminan la visibilidad de las clases, atributos y métodos

3. Clases y objetos

Modificadores de acceso

- Visibilidad de los modificadores de acceso Java:

	La misma clase	Otra clase del mismo paquete	Subclase de otro paquete	Otra clase de otro paquete
<code>public</code>	✓	✓	✓	✓
<code>protected</code>	✓	✓	✓	✗
<code>-</code>	✓	✓	✗	✗
<code>private</code>	✓	✗	✗	✗

Para una clase, los únicos modificadores de acceso permitidos son `public` y `-`. En otras palabras, los modificadores `protected` y `private` solo aplican a métodos y atributos

3. Clases y objetos

Métodos y atributos estáticos

- Se definen mediante el modificador `static`
- Sirve para crear miembros que pertenecen a la clase, y no a una instancia de la clase
- Atributos estáticos:
 - Mantienen información relacionada con el concepto representado por la clase pero se pueden usar sin que tenga que existir ningún objeto de la clase
 - Si se accede desde una clase distinta, debe anteponerse a su nombre el de la clase en la que se encuentra mediante la notación punto
- Métodos estáticos:
 - Operaciones relacionadas con el concepto representado por la clase pero que se pueden invocar sin necesidad de que exista ningún objeto de la clase
 - Igual que los atributos, se invocan mediante la notación punto usando el nombre de la clase seguido de nombre del método
 - Se recomienda usar sólo si son estrictamente necesarios, ya que rompen la filosofía de programación orientada a objetos

3. Clases y objetos

Métodos, atributos, y variables finales

- El modificador `final` indica que una variable, método o clase no se va a modificar
- Esto puede ser útil para añadir más semántica o por cuestiones de rendimiento:
 - Si una variable se marca como `final`, no se podrá asignar un nuevo valor a la variable (constante)
 - Si una clase se marca como `final`, no se podrá extender la clase
 - Si es un método el que se declara como `final`, no se podrá sobrescribir

3. Clases y objetos

Creación de objetos

- Un objeto es una instancia de una clase
- Se utiliza el operador `new`

```
Point point = new Point(1.0, 2.0);
```

- En Java no se declaran destructores porque no es necesario liberar la memoria de forma explícita (el recolector de basura libera la memoria de los objetos cuando no hay ninguna referencia a ellos en el programa)
- Para pedir algo al objeto, hay que hacerlo a través de sus métodos

```
double x = point.getX();  
point.setY(3.0);
```

3. Clases y objetos

Reglas de estilo

- **Clases:** primera letra de cada palabra en mayúsculas
 - Ejemplo: `ClassName`
- **Métodos:** primera letra minúscula y primera letra de cada palabra siguiente con mayúscula (*camelCase*)
 - Ejemplo: `methodName`
- **Variables:** primera letra minúscula y primera letra de cada palabra siguiente con mayúscula (como los métodos)
 - Ejemplo: `variableName`
- **Constantes:** todas en mayúsculas, separando palabras por guion bajo `_`
 - Ejemplo: `CONSTANT_NAME`
- **Paquetes:** en minúsculas, nombre de dominio al revés. Se desaconseja el paquete anónimo
 - Ejemplo: `com.utad.degree.course`

Buenas prácticas,
no obligatorias

3. Clases y objetos

Documentación

- Los comentarios nos ayudan a añadir información extra al código Java (es importante tener un equilibrio en cuánto comentarios)
- Los comentarios en código Java se tienen que reservar para situaciones poco evidentes (por lo tanto no debería haber demasiados comentarios)
- La documentación JavaDoc se puede hacer a nivel de clase, método, o atributo
- A veces se usan las palabras TODO (*por hacer*) y FIXME (*arréglame*) en los comentarios

```
// TODO: This method is not implemented yet  
// FIXME: This method is not correctly implemented
```

3. Clases y objetos

Documentación

Tag	Descripción	Ejemplo
<code>@author</code>	Nombre desarrollador	<code>@author Name Surname</code>
<code>@version</code>	Versión de la clase o método	<code>@version 1.0.0</code>
<code>@param</code>	Parámetro y descripción	<code>@param px Description</code>
<code>@return</code>	Descripción de lo que devuelve un método	<code>@return Description</code>
<code>@throws</code>	Tipo de excepción y descripción lanzada por un método	<code>@throws Exception Description</code>
<code>@see</code>	Relación con otro método o clase, e incluso una URL	<code>@see #method() @see class#method() @see package.class @see package.class#method()</code>

3. Clases y objetos

Documentación

Ejemplo de
JavaDoc de la
clase de
ejemplo `Point`

The screenshot shows the JavaDoc documentation for the `Point` class. The interface includes a navigation bar with 'PACKAGE', 'CLASS', 'USE', 'TREE', 'DEPRECATED', 'INDEX', and 'HELP'. Below this, there are links for 'PREV CLASS', 'NEXT CLASS', 'FRAMES', and 'NO FRAMES', and a 'SUMMARY' section with links for 'NESTED', 'FIELD', 'CONSTR', 'METHOD', and 'DETAIL: FIELD | CONSTR | METHOD'. The main content area displays the package `com.utad.idcd.redes` and the class name **Class Point**. It shows the inheritance hierarchy: `java.lang.Object` and `com.utad.idcd.redes.Point`. The class declaration is shown as `public class Point extends java.lang.Object`. A description states: 'Cartesian plane point (X,Y)'. The version is listed as '1.0.0'. Below this, there is a 'Constructor Summary' section with a 'Constructors' tab. It lists two constructors: `Point()` and `Point(double px, double py)`. A 'Method Summary' section follows, with tabs for 'All Methods', 'Static Methods', 'Instance Methods', and 'Concrete Methods'. The 'All Methods' tab is active, showing a table with columns 'Modifier and Type' and 'Method and Description'. The methods listed are: `double distance(Point target)`, `double getX()`, `double getY()`, `static void main(java.lang.String[] args)`, and `void setX(double x)`.

3. Clases y objetos

La API de Java

- Bibliotecas estándar de clases
 - Clases bien documentadas
 - Organizadas en paquetes
- El conocimiento de la API estándar de Java es imprescindible para cualquier desarrollador Java
- Antes de intentar resolver un problema del programa que estamos abarcando
 - Reflexionar qué paquetes son necesarios para resolverlo, o si ya está resuelto

3. Clases y objetos

La API de Java

- **Algunos paquetes:**
 - `java.lang`: incluye las clases del lenguaje Java propiamente dicho: `Object`, `Thread`, `Exception`, `System`, `Integer`, `Float`, `Math`, `String`, etc
 - `java.io`: El paquete de entrada/salida contiene las clases de acceso a ficheros: `FileInputStream` y `FileOutputStream`
 - `java.util`: conjunto de clases útiles como: `Date` (fecha), `Dictionary` (diccionario), `Random` (números aleatorios) y `Stack` (pila FIFO)
 - `java.net`: Clases para aplicaciones de red. Incluye sockets TCP y UDP

3. Clases y objetos

La API de Java

- `java.lang.Object` es la raíz de la jerarquía de clases de Java. Proporciona métodos de utilidad general que pueden utilizar todos los objetos:
 - Convertir a un `String` las características de estado de un objeto:
`object.toString();`
 - Devolver la clase del objeto:
`object.getClass();`
 - Comparar un objeto con otro:
`object.equals(object2);`

3. Clases y objetos

La API de Java

- `java.lang.Math` es una clase con funciones matemática en Java
- No es instanciable (sus métodos están definidos como estáticos)
- Define algunas constantes: `Math.E` = número e, `Math.PI` = número pi

Retorno	Función	Descripción
double	<code>Math.sin(double a)</code>	Seno en radianes del ángulo a
double	<code>Math.cos(double a)</code>	Coseno en radianes del ángulo a
double	<code>Math.tan(double a)</code>	Tangente en radianes del ángulo a
double	<code>Math.exp(double x)</code>	e elevado a x
double	<code>Math.log(double x)</code>	Logaritmo de x
double	<code>Math.sqrt(double x)</code>	Raíz cuadrada de x
double	<code>Math.pow(double x, double y)</code>	Devuelve y elevado a x
(int, long, float, double)	<code>Math.max(a, b)</code> <code>Math.min(a, b)</code>	Devuelve el máximo o el mínimo (para int, long, float, double)

3. Clases y objetos

La API de Java

- `java.lang.String` es la clase Java para cadenas de caracteres

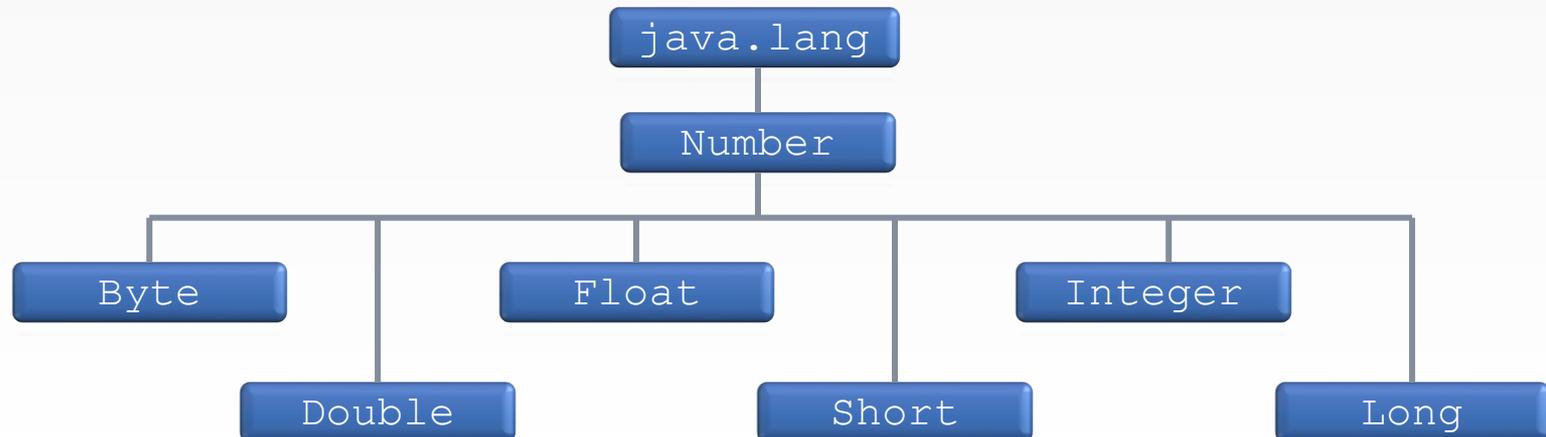
Retorno	Función	Descripción
int	<code>length()</code>	Número de caracteres
char	<code>charAt(int p)</code>	Devuelve el carácter en la posición especificada
int	<code>indexOf(String s, int p)</code>	Posición en la que aparece por primera vez un <code>String</code> en otro <code>String</code> , a partir de una posición
int	<code>indexOf(int c, int p)</code>	Primera ocurrencia del carácter <code>c</code> a partir de la posición <code>p</code>
String	<code>replace(char i, char f)</code>	Sustituye un carácter <code>i</code> por otro <code>f</code>
String	<code>substring(int i, int f)</code>	Devuelve un <code>String</code> extraído de otro desde <code>i</code> hasta <code>f</code>
String	<code>toLowerCase()</code>	Convierte en minúsculas
String	<code>toUpperCase()</code>	Convierte en mayúsculas
String	<code>trim()</code>	Elimina los espacios en blanco al comienzo y final de la cadena
String	<code>valueOf(Object obj)</code>	Devuelve la representación en <code>String</code> del objeto <code>obj</code> (también puede ser <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code>)

3. Clases y objetos

La API de Java

- *Wrappers*

- Clases envoltorio diseñadas como complemento a los tipos primitivos:
 - Conversor entre tipos (de `String` a entero, etc.)
 - Tratamiento de tipos básicos como objetos
- Contienen una variable de tipo primitivo, que es la que se quiere modificar
- Existe una clase *wrapper* por cada tipo primitivo:



3. Clases y objetos

La API de Java

- *Wrapper*
 - Las clases envoltorio siempre tienen la primera letra en mayúscula:
 - Byte **para** byte
 - Short **para** short
 - Integer **para** int
 - Long **para** long
 - Boolean **para** boolean
 - Float **para** float
 - Double **para** double
 - **También:** Character **para** char

3. Clases y objetos

La API de Java

- El *wrapper* Integer

Retorno	Función	Descripción
Constructores	<code>Integer(int value)</code>	Construye un <code>Integer</code> con el valor <code>int</code> especificado
	<code>Integer (String s)</code>	Construye un <code>Integer</code> con el valor <code>int</code> indicado en el <code>String s</code>
<code>boolean</code>	<code>equals (Obj obj)</code>	Compara este objeto con el especificado en <code>obj</code>
<code>int</code>	<code>intValue()</code>	Devuelve el valor <code>int</code> del <code>Integer</code>
<code>static int</code>	<code>parseInt (String s)</code>	Convierte a entero el número contenido en un <code>String</code>
<code>String</code>	<code>toString()</code>	Convierte a <code>String</code> el contenido del <code>Integer</code>
<code>static String</code>	<code>toString(int i)</code>	Devuelve un <code>String</code> con el entero especificado
<code>static Integer</code>	<code>valueOf (String s)</code>	Devuelve un <code>Integer</code> con el valor del <code>String</code> especificado

3. Clases y objetos

La API de Java

- `java.lang.System` es una clase Java proporciona mecanismos de entrada/salida:
 - `System.in` = entrada estándar
 - `System.out` = salida estándar
 - `System.err` = salida error estándar
- Para escribir en pantalla (salida estándar):

```
System.out.println("This is going to appear in the standard output");
```

- Para leer del teclado (entrada estándar) podemos utilizar la clase `Scanner` (paquete `java.util`)

3. Clases y objetos

La API de Java

- `java.util.Scanner` es una clase que facilita la lectura de datos desde teclado:

Retorno	Función	Descripción
Constructor	<code>Scanner(String s)</code>	Construye un Scanner con un String
boolean	<code>nextBoolean()</code>	Lee un valor de tipo boolean
double	<code>nextDouble()</code>	Lee un valor de tipo double
float	<code>nextFloat()</code>	Lee un valor de tipo float
int	<code>nextInt()</code>	Lee un valor de tipo int
String	<code>nextLine()</code>	Lee una línea de tipo String
void	<code>close()</code>	Cierra el flujo de entrada

Índice de contenidos

1. Introducción a Java
2. Elementos básicos
3. Clases y objetos
4. Elementos avanzados
 - Comparación de variables
 - Paso de parámetros
 - Anotaciones
 - Herencia
 - Interfaces
 - Clases abstractas
 - Gestión de excepciones
 - Tipos enumerados
 - Clases internas
5. Introducción a Eclipse
6. Gestión de entrada/salida
7. Colecciones
8. Programación con sockets
9. Hilos en Java
10. Para ampliar

4. Elementos avanzados

Comparación de variables

- Como hemos visto, las variables en Java puedes almacenar tipos primitivos o referencias a objetos. Por ejemplo:

```
int i1 = 10;
```

```
String s = "hello";
```

- El contenido de las variables se almacena en la zona de memoria *stack*, mientras que los objetos se almacenan en el *heap*
- Para comparar variables definidas con tipos primitivos usamos el operador `==`, mientras que para comparar objetos usamos el método `equals` (heredado de la clase `Object`)

4. Elementos avanzados

Comparación de variables

```
public class PrimitiveTest {  
  
    public static void main(String[] args) {  
        int i1 = 10;  
        int i2 = 100;  
  
        System.out.println(i1 == i2); // the output is "false"  
    }  
  
}
```

s1

10

s2

100

La igualdad de variables de tipos primitivas se produce en base al valor de las mismas

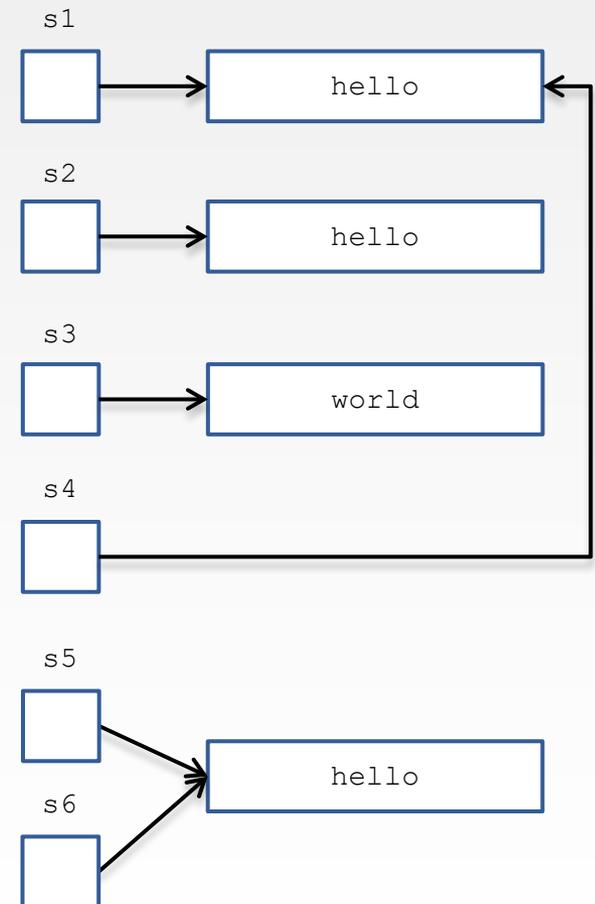
4. Elementos avanzados

Comparación de variables

```
String s1 = new String("hello");
String s2 = new String("hello");
String s3 = new String("world");
String s4 = s1;
String s5 = "hello";
String s6 = "hello";

System.out.println(s1.equals(s2)); // the output is "true"
System.out.println(s1.equals(s3)); // the output is "false"
System.out.println(s1 == s2); // the output is "false"
System.out.println(s1 == s4); // the output is "true"
System.out.println(s1 == s5); // the output is "false"
System.out.println(s5 == s6); // the output is "true"
```

Las cadenas definidas con = se almacenan en un espacio de memoria dentro del heap llamado *String pool*. Por eficiencia, en esta zona, las cadenas de iguales contenido apuntan a la misma zona de memoria (esto se conoce como “*String interning*”)



4. Elementos avanzados

Comparación de variables

```
public class Person {  
  
    private String name;  
    private String surname;  
  
    public Person(String name, String surname) {  
        this.name = name;  
        this.surname = surname;  
    }  
  
    // Getters and setters ...  
  
    @Override  
    public boolean equals(Object arg0) {  
        return ((Person) arg0).getName().equals(this.getName())  
            && ((Person) arg0).getSurname().equals(this.getSurname());  
    }  
}
```

```
public class PersonTest01 {  
  
    public static void main(String[] args) {  
        Person p1 = new Person("John", "Smith");  
        Person p2 = new Person("Michael", "Jordan");  
        Person p3 = new Person("Michael", "Jordan");  
  
        System.out.println(p1.equals(p2)); // false  
        System.out.println(p2.equals(p3)); // true  
    }  
}
```

El método `equals` se puede sobrescribir en cualquier clase para realizar una comparación en base a los atributos de la clase

4. Elementos avanzados

Paso de parámetros

- El paso de parámetro en Java es siempre **por valor**

```
public class PrimitiveTest02 {  
  
    public static void main(String[] args) {  
        int i1 = 10;  
        System.out.println(i1);  
        changeInt(i1, 100);  
        System.out.println(i1);  
    }  
  
    private static void changeInt(int i, int newInt) {  
        i = newInt;  
    }  
  
}
```

i1

10

Salida por pantalla:

10
10

4. Elementos avanzados

Paso de parámetros

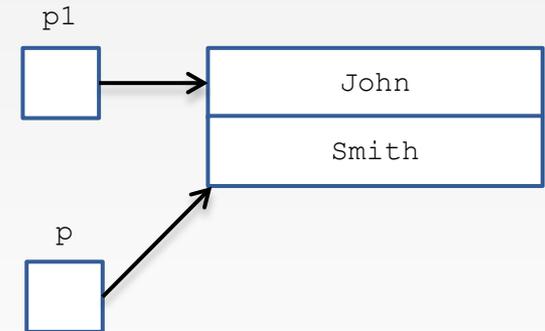
- En el paso de variables que referencian objetos, se copia la referencia:

```
public class PersonTest02 {

    public static void main(String[] args) {
        Person p1 = new Person("John", "Smith");
        System.out.println(p1.getName());
        changeName(p1, "Michael");
        System.out.println(p1.getName());
    }

    private static void changeName(Person p, String newName) {
        p.setName(newName);
    }

}
```



Salida por pantalla:

```
John
Michael
```

4. Elementos avanzados

Paso de parámetros

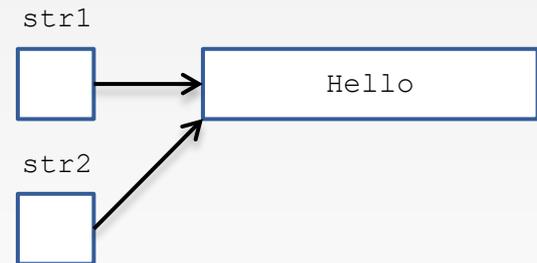
- En el paso de variables que referencian objetos, se copia la referencia:

```
public class StringTest01 {

    public static void main(String[] args) {
        String str1 = "Hello";
        System.out.println(str1);
        changeString(str1);
        System.out.println(str1);
    }

    private static void changeString(String str2) {
        str2 = "Goodbye";
    }

}
```



Salida por pantalla:

```
Hello
Hello
```

4. Elementos avanzados

Anotaciones

- Una anotación Java es una forma de añadir metadatos al código fuente Java que están disponibles para la aplicación en tiempo de ejecución
- Están disponibles en Java desde la versión 5
- Paquetes, clases, atributos, métodos, constructores, parámetros, y variables pueden ser anotados
- Ejemplos:

```
public class AnnotationExample {
    @Deprecated
    public static String attr

    @Override
    public String toString() {
        return "";
    }

    @SuppressWarnings("unused")
    public static void main(String[] args) {
        String a;
    }
}
```

Anotación	Descripción
@Override	Un método sobrescrito de la superclase
@Deprecated	Elemento obsoleto
@SuppressWarnings	Ocultar advertencias en tiempo de compilación

4. Elementos avanzados

Herencia

- Permite crear una nueva clase (**subclase** o clase hija) a partir de otra que ya existe (**superclase** o clase padre) “heredando” sus **atributos** y sus **métodos** (`public` o `protected`)
- Se usa la palabra reservada `extends`

```
public class SubClass extends SuperClass {  
  
}
```

- La subclase, además de heredar los atributos y métodos de la superclase puede:
 - Añadir nuevos atributos o métodos
 - Redefinir métodos de la clase padre
- No existe herencia múltiple en Java (una clase sólo puede tener otra clase como padre)

4. Elementos avanzados

Herencia

- Una subclase hereda todos los **atributos** de su superclase
- Desde la subclase se puede acceder a los atributos declarados como `public` o `protected`
- Redefinición del atributo de superclase: si la subclase declara un atributo con el mismo nombre que otro de la clase padre

```
public class SuperClass {  
    public String string = "superclass";  
}
```

```
public class SubClass extends SuperClass {  
    private String string = "subclass";  
  
    public SubClass() {  
        System.out.println(string);  
        System.out.println(super.string);  
    }  
  
    public static void main(String[] args) {  
        new SubClass();  
    }  
}
```

Salida por pantalla:

```
subclass  
superclass
```

4. Elementos avanzados

Herencia

- Una subclase hereda todos los **métodos** de la superclase
- Desde la subclase se puede acceder a los métodos declarados como `public` o `protected`
- Si una subclase declara un método con el mismo nombre y parámetros que otros método de la superclase, se sobrescribe (redefine) el método de la superclase y se usa el de la subclase
- Métodos que una subclase no puede sobrescribir
 - Métodos declarados como `final` en la superclase
 - Tampoco puede sobrescribir métodos `static`

4. Elementos avanzados

Herencia

- Los constructores se heredan, pero no se pueden utilizar para crear objetos de la subclase
- Al crear un objeto de una subclase se llamará primero al constructor de la superclase (para crear e inicializar lo común)
- Si no se indica nada en el constructor, automáticamente se llamará al constructor sin parámetros de la superclase
- Para indicar a que constructor llamar se emplea: `super()` con los parámetros necesarios
 - Ejemplo: `super(p1, p2)`

```
public class SuperClass {  
    public SuperClass() {  
        System.out.println("Constructor SuperClass");  
    }  
}
```

```
public class SubClass extends SuperClass {  
  
    public SubClass() {  
        System.out.println("Constructor SubClass");  
    }  
  
    public static void main(String[] args) {  
        new SubClass();  
    }  
}
```

Salida por pantalla:

```
Constructor SuperClass  
Constructor SubClass
```

4. Elementos avanzados

Interfaces

- Especifica un conjunto de métodos sin implementar (apariencia de una clase)
- Sirven para separar la especificación (interfaz) de la implementación
- Puede definir atributos constantes (`static final`)
- Los interfaces se especifican usando la palabra reservada `interface`
- Esos métodos serán implementados en otra clase que “implemente el interfaz” (`implements`)
- Una clase no puede heredar de varias clases, pero sí puede implementar varias interfaces

4. Elementos avanzados

Interfaces

```
public interface Figure {  
  
    public double PI = Math.PI;  
  
    public double area();  
}
```

```
public class Circulo implements Figure {  
  
    private double radius;  
  
    public Circulo(double radius) {  
        this.radius = radius;  
    }  
  
    public double area() {  
        return (PI * radius * radius);  
    }  
}
```

```
public class Square implements Figure {  
  
    private double side;  
  
    public Cuadrado(double side) {  
        this.side = side;  
    }  
  
    public double area() {  
        return side * side;  
    }  
}
```

4. Elementos avanzados

Clases abstractas

- Las clases abstractas son el termino medio entre clases normales e interfaces, ya que definen la apariencia de una clase (métodos abstractos) pero puede contener también métodos implementados
- Un método abstracto (método sin implementación) se declara utilizando la palabra reservada `abstract`
- Una clase que tiene un método abstracto debe ser declarada obligatoriamente como abstracta mediante la palabra reservada `abstract`
- No se pueden crear objetos de una clase abstracta
- Una clase abstracta necesita una clase que herede de ella (con `extends`). Para que se pueda instanciar debe implementar todos los métodos abstractos.

4. Elementos avanzados

Clases abstractas

```
public abstract class Instrument {  
    public abstract void play();  
  
    public void tune() {  
        // ...  
    }  
}
```

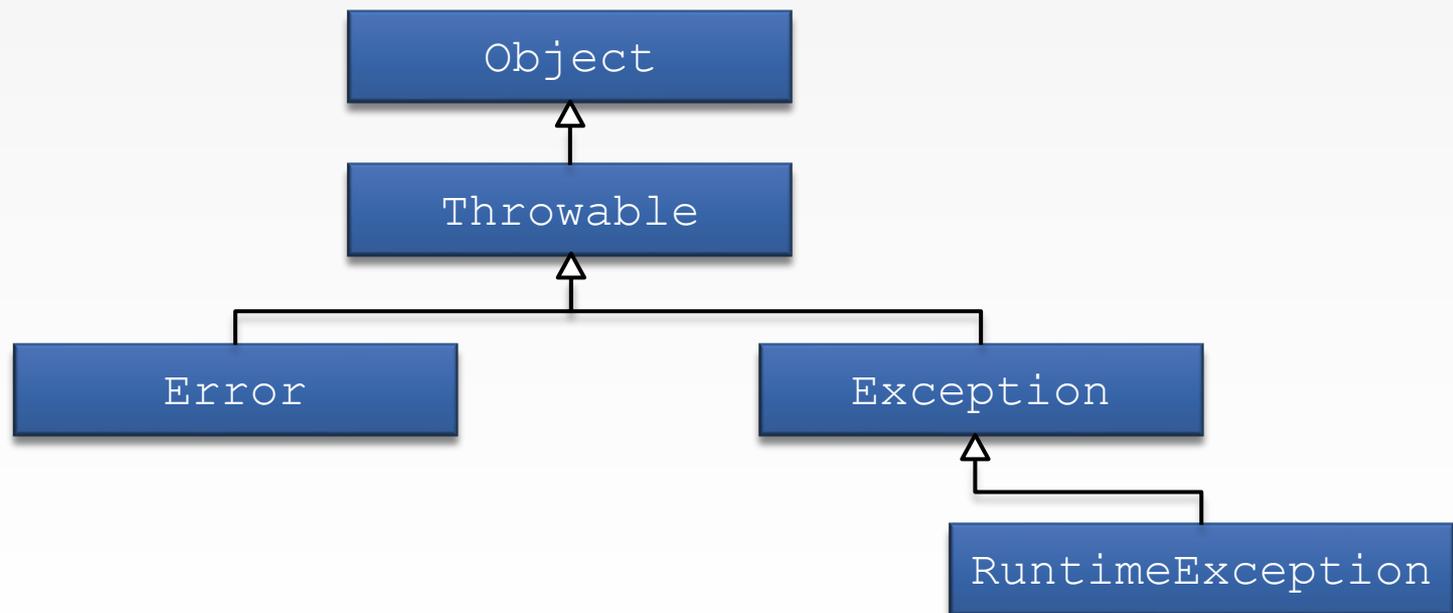
```
public class Guitar extends Instrument {  
  
    @Override  
    public void play() {  
        // ...  
    }  
}
```

```
public class Drums extends Instrument {  
  
    @Override  
    public void play() {  
        // ...  
    }  
}
```

4. Elementos avanzados

Gestión de excepciones

- Una excepción es un evento que ocurre durante la ejecución de un programa e interrumpe el flujo normal de ejecución
- Un objeto de alguna clase derivada de `java.lang.Throwable`:



4. Elementos avanzados

Gestión de excepciones

- Errores
 - Clases derivadas de `java.lang.Error`
 - Problemas de la máquina virtual
- Excepciones
 - Clases derivadas de `java.lang.Exception`
 - Problemas normales de una aplicación Java
 - La excepción ha de ser tratada
- Excepciones no chequeadas
 - Clases derivadas de `java.lang.RuntimeException`
 - No es necesario tratar las excepciones de este tipo

4. Elementos avanzados

Gestión de excepciones

- **Ejemplos de excepciones:**
 - `java.io.IOException`: **Problema genérico de entrada/salida**
 - `java.net.MalformedURLException`: **Uso de URL inválida**
 - `java.io.FileNotFoundException`: **Fichero no encontrado**
 - `java.lang.NoSuchMethodException`: **Método no encontrado**
- **Ejemplos de excepciones no chequeadas:**
 - `java.lang.NullPointerException`: **Acceso o modificación de un campo a valor null**
 - `java.lang.IndexOutOfBoundsException`: **Uso de un índice fuera de rango (por ejemplo, de un array)**
 - `java.lang.NumberFormatException`: **Error al convertir cadena a número**

4. Elementos avanzados

Gestión de excepciones

- ¿Cómo tratar una excepción? Hay dos alternativas:
 1. Capturar la excepción (mediante `try-catch-finally`)
 - El bloque `finally` es opcional

Salida del programa sin argumentos

```
import java.net.MalformedURLException;
import java.net.URL;

public class ExceptionsExample02 {

    public static void main(String[] args) {
        URL utad = null;
        try {
            utad = new URL(args[0]);
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }

        if (utad != null) {
            System.out.println(utad.getHost());
        }
    }
}
```

Salida del programa con argumentos:
http://www.u-tad.com

www.u-tad.com

Salida del programa con argumentos: aaaaa

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
at ExceptionsExample02.main(ExceptionsExample02.java:9)
```

```
java.net.MalformedURLException: no protocol: aaaaa
at java.net.URL.<init>(URL.java:586)
at java.net.URL.<init>(URL.java:483)
at java.net.URL.<init>(URL.java:432)
at ExceptionsExample02.main(ExceptionsExample02.java:9)
```

4. Elementos avanzados

Gestión de excepciones

- ¿Cómo tratar una excepción? Hay dos alternativas:
 2. Propagar la excepción (mediante `throws`)

```
import java.net.MalformedURLException;
import java.net.URL;

public class ExceptionsExample01 {

    public static void main(String[] args) throws MalformedURLException {
        URL utad = new URL("http://www.u-tad.com/");
    }
}
```

4. Elementos avanzados

Gestión de excepciones

- Una excepción se puede lanzar mediante la palabra reservada `throw`:

```
public static void main(String[] args) throws Exception {  
    if (args.length < 1) {  
        throw new Exception("Invalid number of arguments");  
    }  
}
```

```
public static void main(String[] args) {  
    if (args.length < 1) {  
        throw new RuntimeException("Invalid number of arguments");  
    }  
}
```

4. Elementos avanzados

Tipos enumerados

- Tipo especial de datos que permite que una variable tenga valor dentro un conjunto de valores predefinidos

```
public enum Day {  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;  
}
```

```
public class EnumExample {  
    public static void main(String[] args) {  
        Day day = Day.SATURDAY;  
        switch (day) {  
            case MONDAY:  
                System.out.println("Mondays are bad");  
                break;  
            case FRIDAY:  
                System.out.println("Fridays are better");  
                break;  
            case SATURDAY:  
            case SUNDAY:  
                System.out.println("Weekends are the best");  
                break;  
            default:  
                System.out.println("Midweek days are so-so");  
                break;  
        }  
    }  
}
```

4. Elementos avanzados

Clases internas

- Es posible embeber clases dentro de otras clases
- Son las llamadas “clases internas” o “clases anidadas” (*inner classes* o *nested classes*)
- Hay que usarlas con precaución, ya que pueden romper el buen diseño de un sistema orientado a objetos

```
public class Outer {  
  
    public void sayHello() {  
        System.out.println("Hello from the main class");  
    }  
  
    public void callInner() {  
        Inner inner = new Inner();  
        inner.sayHello();  
    }  
  
    public static void main(String[] args) {  
        Outer outer = new Outer();  
        outer.sayHello();  
        outer.callInner();  
    }  
  
    class Inner {  
        public void sayHello() {  
            System.out.println("Hello from the inner class");  
        }  
    }  
}
```

Índice de contenidos

1. Introducción a Java
2. Elementos básicos
3. Clases y objetos
4. Elementos avanzados
- 5. Introducción a Eclipse**
6. Gestión de entrada/salida
7. Colecciones
8. Programación con sockets
9. Hilos en Java
10. Para ampliar

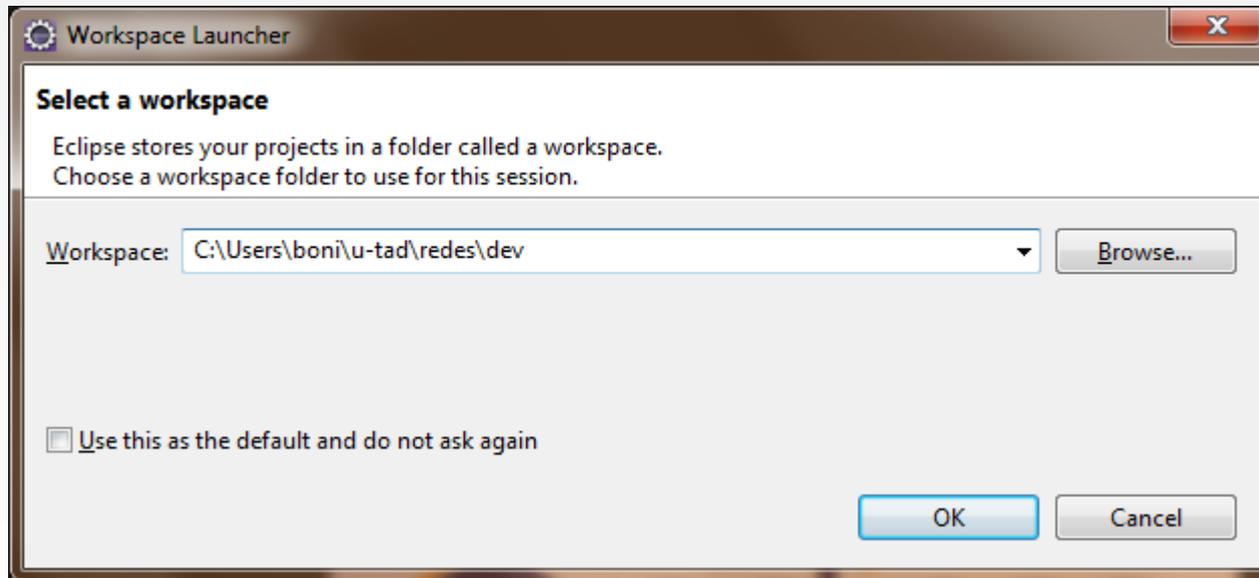
5. Introducción a Eclipse

- Entorno de Desarrollo Integrado (IDE) libre (*open source*)
- A parte de Java, puede ser usado para desarrollar aplicaciones de otras tecnologías:
 - Ada, ABAP, C, C++, COBOL, Fortran, Haskell, JavaScript, Lasso, Natural, Perl, PHP, Prolog, Python, R, Ruby, Scala, Clojure, Groovy, Scheme, Erlang



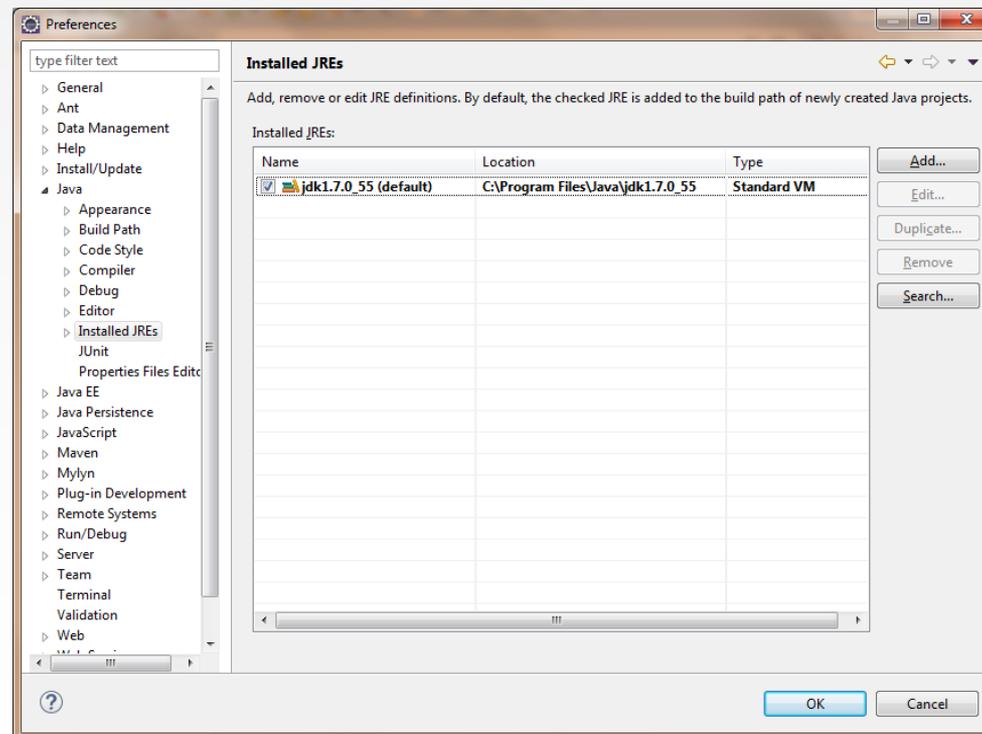
5. Introducción a Eclipse

- Al arrancar Eclipse nos pedirá un directorio de trabajo (*workspace*)



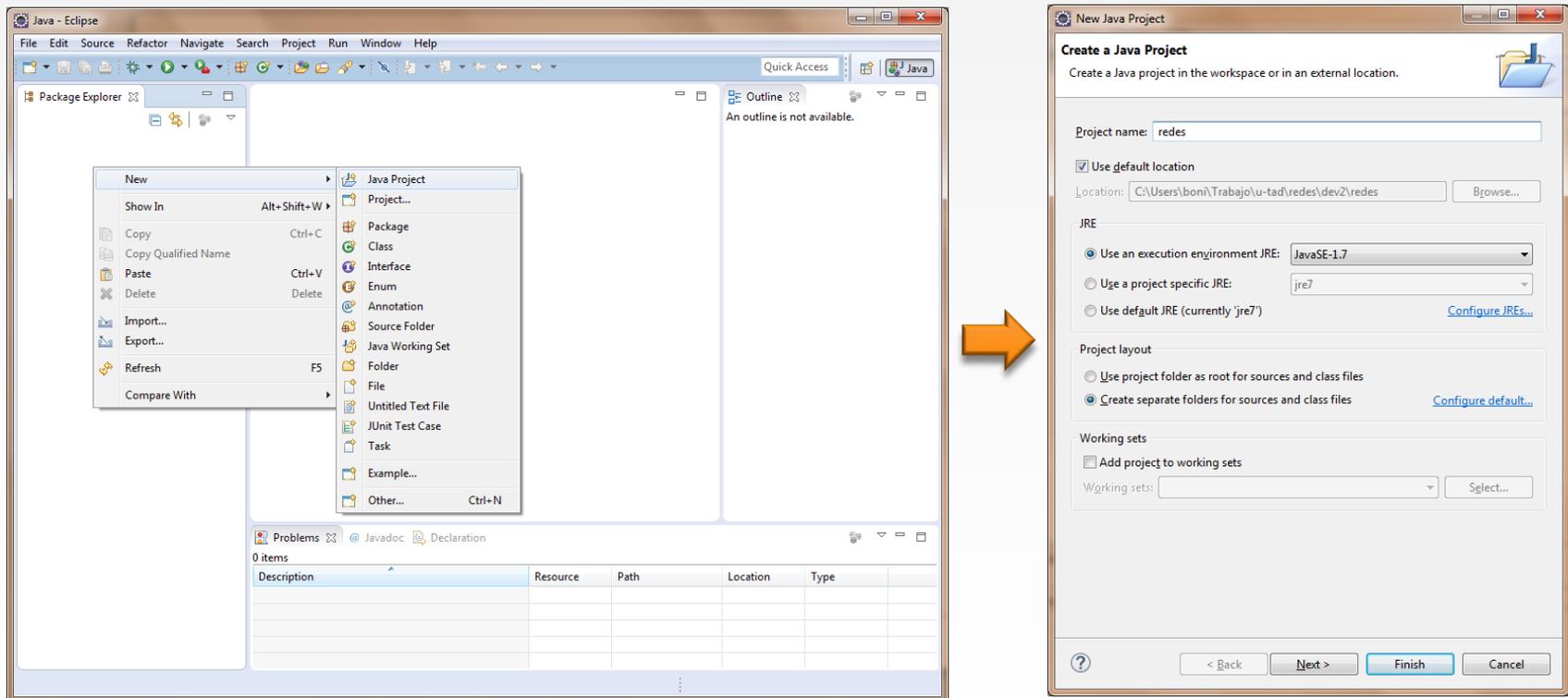
5. Introducción a Eclipse

- Deberá haber al menos una máquina virtual Java (JRE o JDK) instalado y configurado en Eclipse:



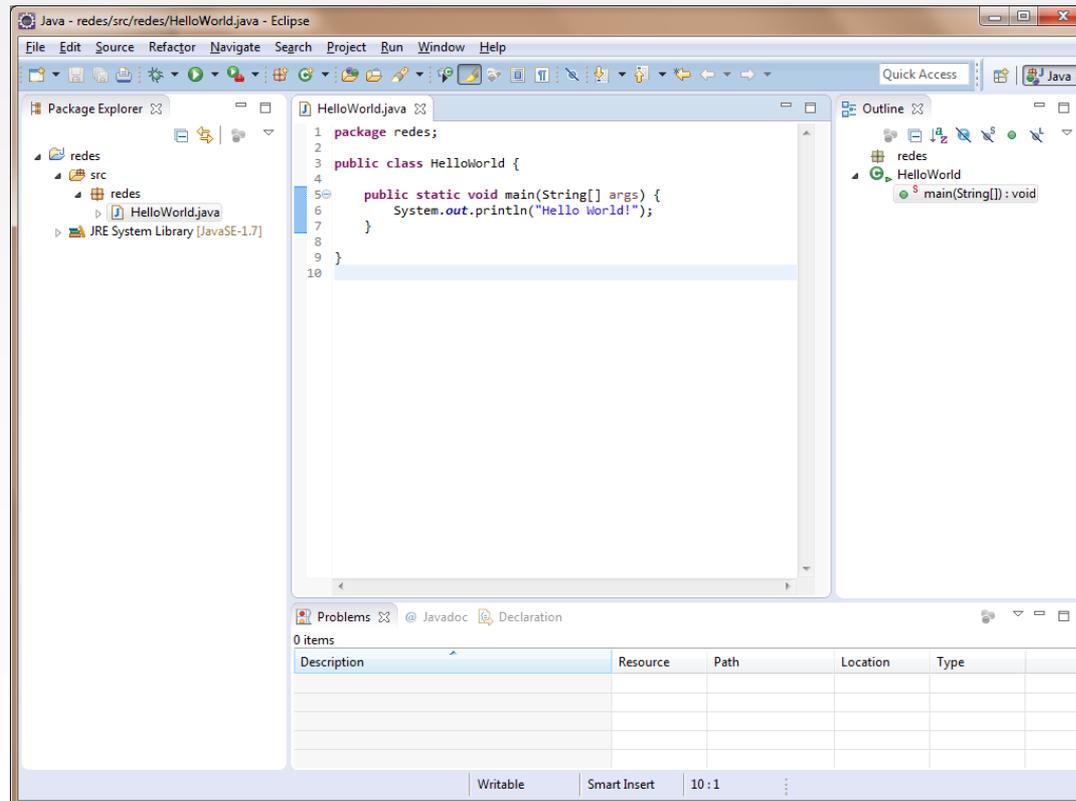
5. Introducción a Eclipse

- El primer paso para crear una aplicación Java será crear un nuevo **proyecto Java**:



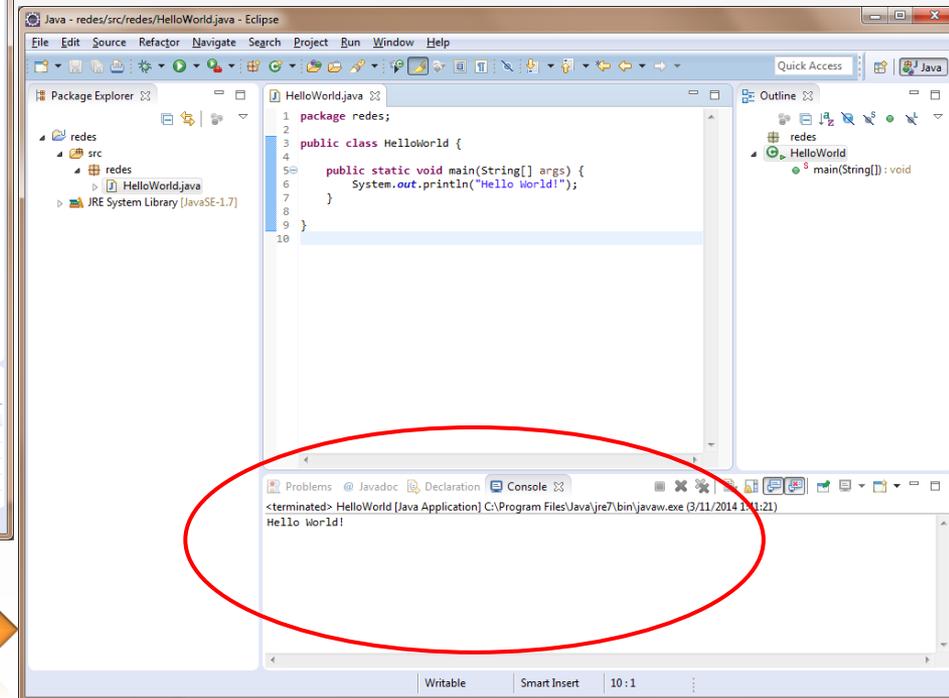
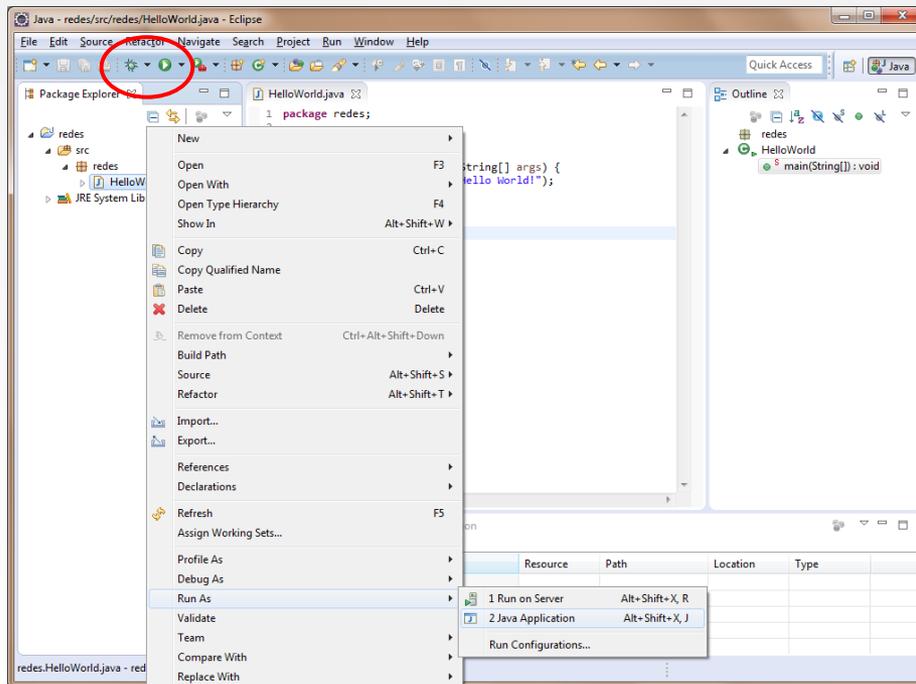
5. Introducción a Eclipse

- “Hola Mundo” en Eclipse:



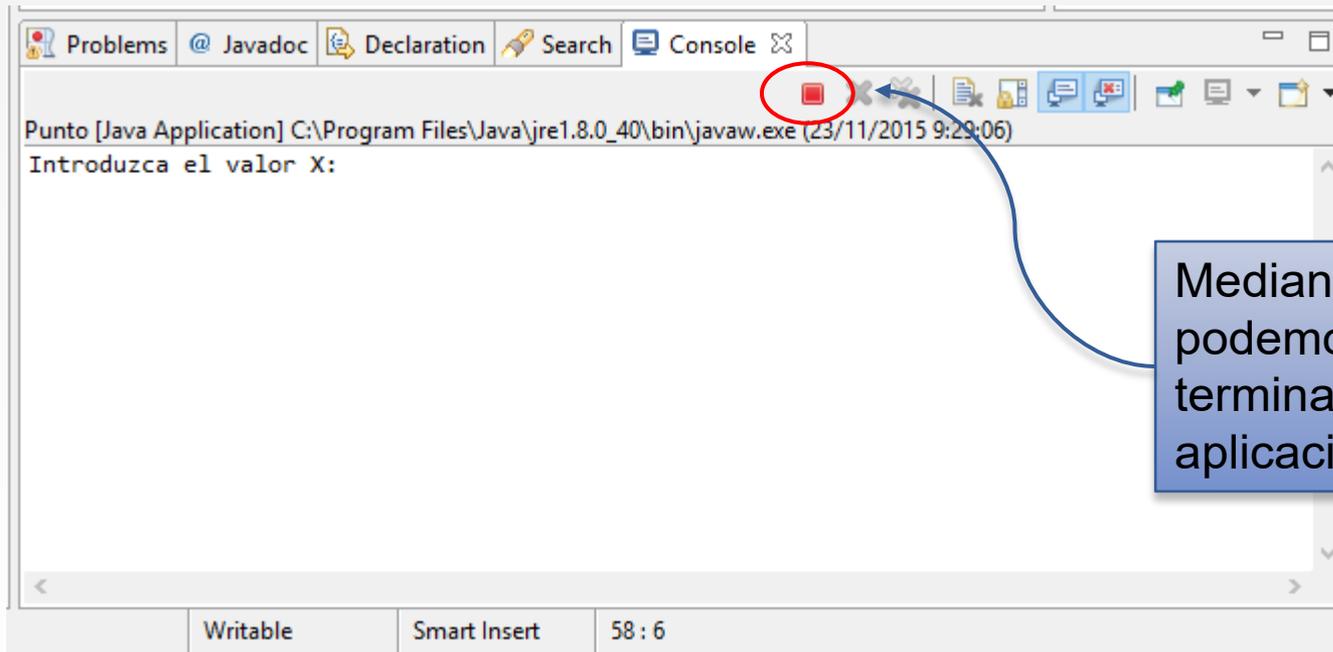
5. Introducción a Eclipse

- Ejecución de “Hello World” en Eclipse:



5. Introducción a Eclipse

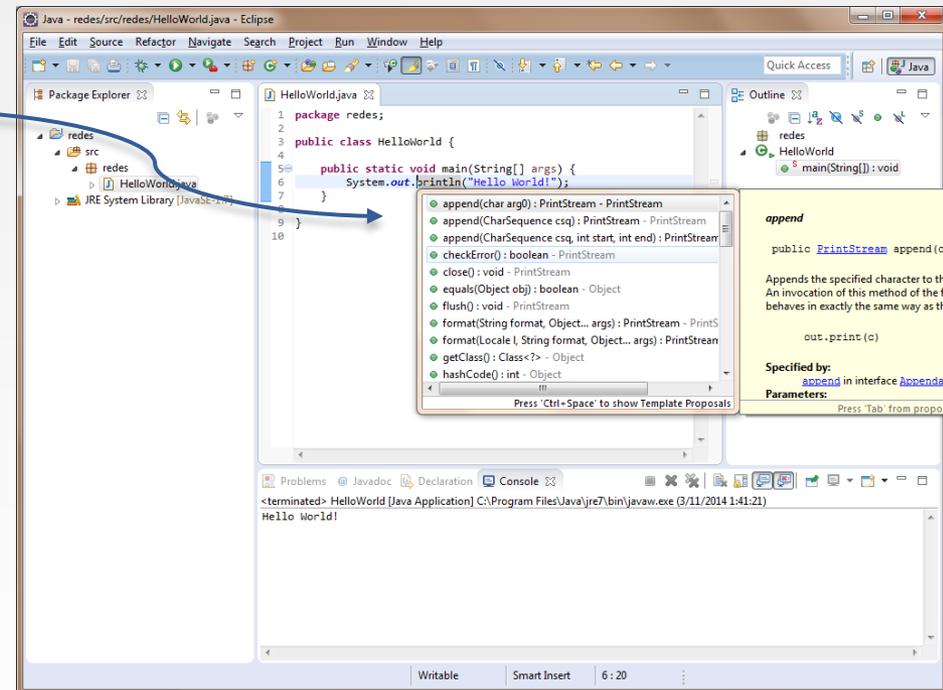
- En la pestaña “*Console*” podemos ver la entrada/salida estándar de nuestra aplicación



Mediante este botón podemos forzar la terminación de una aplicación en ejecución

5. Introducción a Eclipse

- Algunos atajos en Eclipse:
 - Ctrl+Espacio: Autocompletar
 - Ctrl+Shift+O: Organizar importación de paquetes
 - Ctrl+Shift+F: Formatear código
 - Alt+Shift+R: Refactorizar (cambiar nombre de clase, variable, método)



Índice de contenidos

1. Introducción a Java
2. Elementos básicos
3. Clases y objetos
4. Elementos avanzados
5. Introducción a Eclipse
6. Gestión de entrada/salida
 - Java IO
 - Entrada/salida estándar
 - Java NIO
7. Colecciones
8. Programación con sockets
9. Hilos en Java
10. Para ampliar

6. Gestión de entrada/salida

- La API de Java proporciona varios métodos de entrada/salida (E/S)
- Ya hemos visto que la clase `java.lang.System` proporciona mecanismos de E/S sobre el terminal:
 - `System.in` = entrada estándar
 - `System.out` = salida estándar
 - `System.err` = salida error estándar
- En esta sección vamos a ver las clases de E/S de los paquetes:
 - `java.io`: *Input/output* (I/O)
 - `java.nio`: New I/O, desde Java 4
- La I/O en Java está basada en flujos de datos (*streams*), que están ligados a un dispositivo físico

6. Gestión de entrada/salida

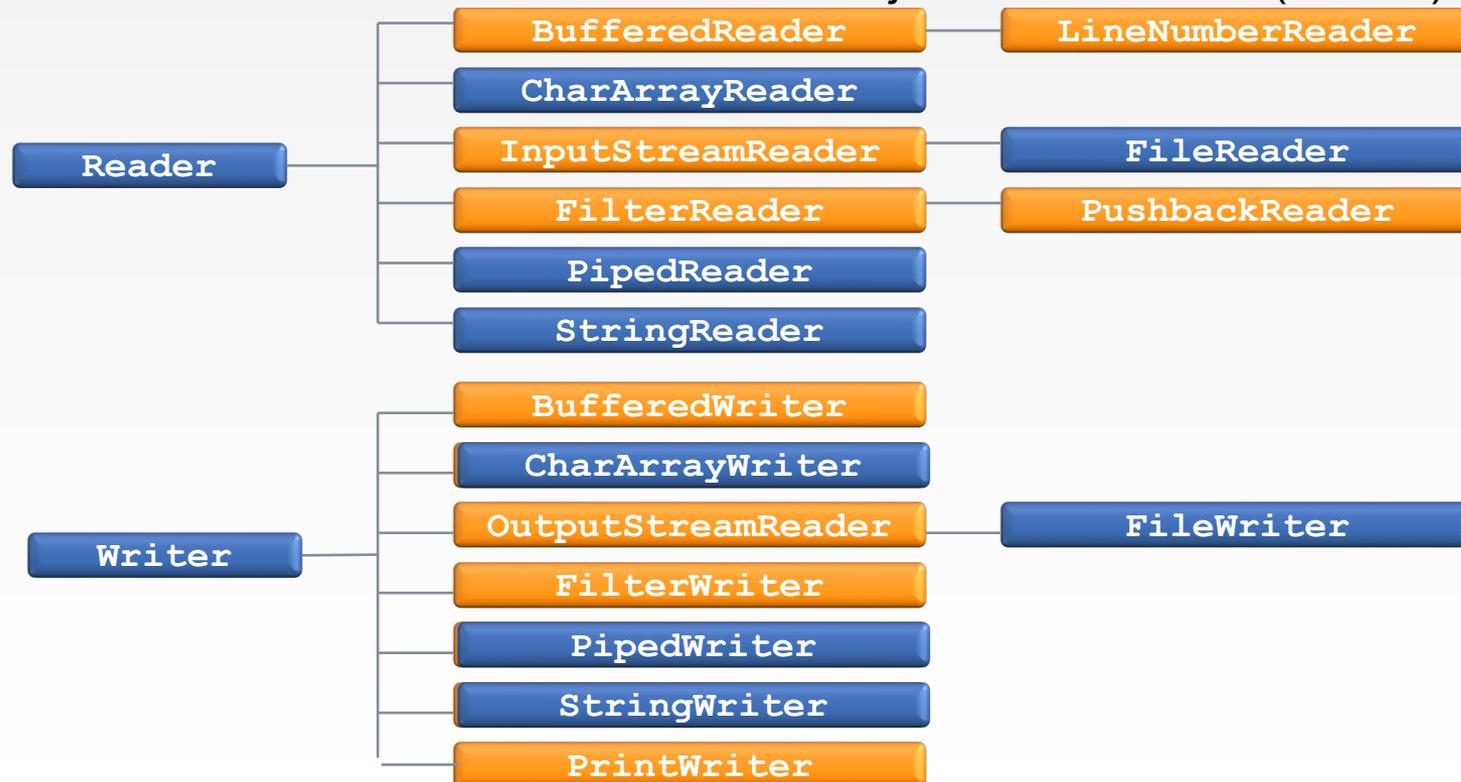
- Hay dos tipos de **flujos de datos** (*streams*):
 - Flujos de caracteres (Clases `java.io.Reader` y `java.io.Writer`): leen o escriben de caracteres (de 16 en 16 bits, Unicode)
 - Flujos de bytes (Clases `java.io.InputStream` y `java.io.OutputStream`): leen o escriben bytes (de 8 en 8 bits)
- Ambos tipos de flujos tienen métodos de lectura y escritura:
 - `Reader` e `InputStream`: tienen métodos de lectura (métodos `read`)
 - `Writer` y `OutputStream`: métodos de escritura (métodos `write`)
- Existen además otras clases, que son las **clases filtro**, que añaden características particulares a la forma de leer/escribir los datos
- El método general consiste en enlazar un flujo y un filtro, creando un canal de comunicación

6. Gestión de entrada/salida

 = Fuente datos
 = Clases filtro

Jerarquía de clases de flujos de caracteres

- `java.io.Reader`: ofrece lectores de flujos de caracteres (16 bits)
- `java.io.Writer`: ofrece escritores de flujos de caracteres (16 bits)

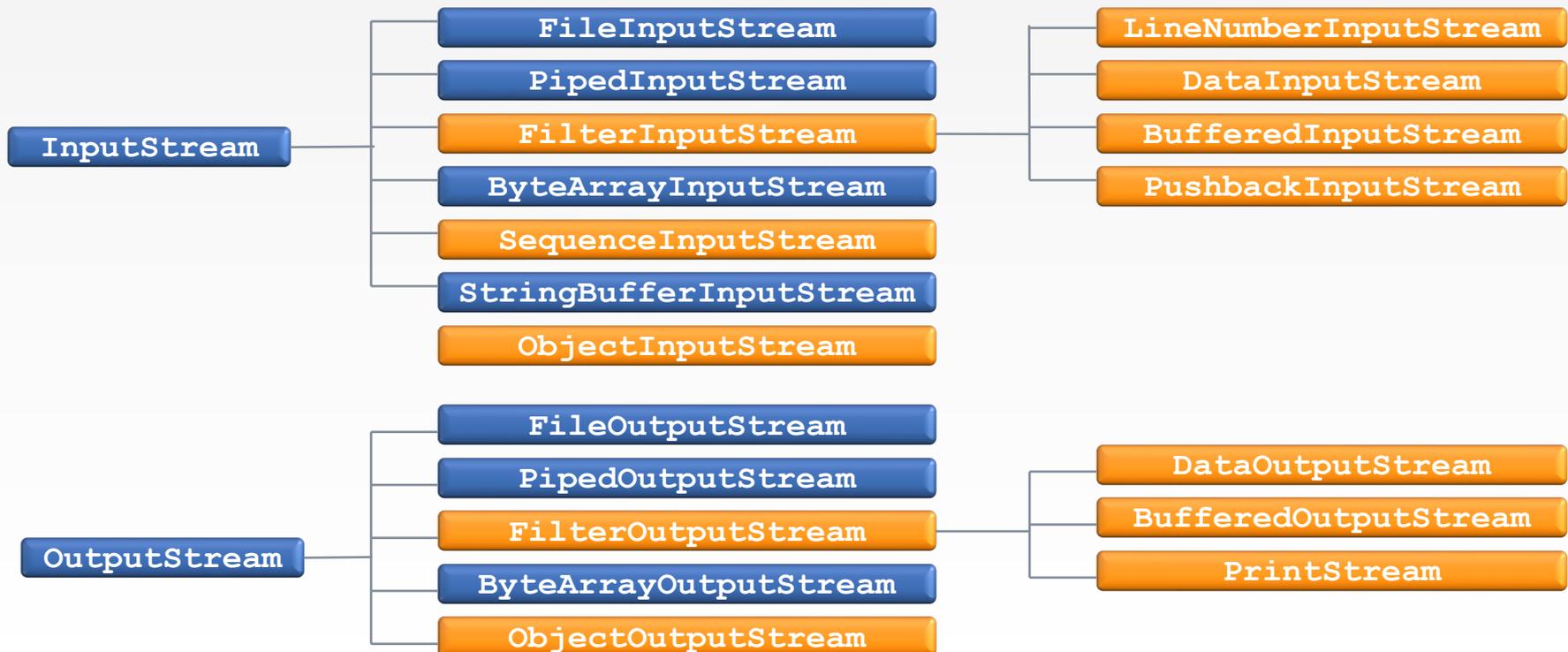


6. Gestión de entrada/salida

■ = Fuente datos
■ = Clases filtro

Flujos de bytes (información no textual)

- `java.io.InputStream`: ofrece lectores de flujos de bytes (8 bits)
- `java.io.OutputStream`: ofrece escritores de flujos de bytes (8 bits)



6. Gestión de entrada/salida

Java IO

- Receta #1. Lectura de un fichero de texto línea por línea:

```
try {
    BufferedReader reader = new BufferedReader(new FileReader("file.txt"));
    String line;
    while ((line = reader.readLine()) != null) {
        // ...
    }
    reader.close();
} catch (FileNotFoundException e) {
    System.err.println("File does not exist: " + e.getMessage());
} catch (IOException e) {
    System.err.println("I/O error: " + e.getMessage());
}
```

6. Gestión de entrada/salida

Java IO

- Receta #2. Escritura en un fichero de texto línea por línea:

```
String[] lines = { "line1", "line2" };  
try {  
    PrintWriter writer = new PrintWriter(new FileWriter("output.txt"));  
    writer.println(lines[0]);  
    writer.println(lines[1]);  
    writer.close();  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

6. Gestión de entrada/salida

Java IO

- Receta #3. Anexar en un fichero de texto línea por línea:

```
String[] lines = { "line1", "line2" };  
try {  
    PrintWriter writer = new PrintWriter(new FileWriter("output.txt", true));  
    writer.println(lines[0]);  
    writer.println(lines[1]);  
    writer.close();  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

6. Gestión de entrada/salida

Java IO

- Clase `java.io.File`: se utiliza para ficheros y directorios

Método	Descripción
<code>File(String path)</code>	Crea una instancia a fichero o directorio
<code>boolean exist()</code>	Investiga si existe un fichero o directorio
<code>boolean delete()</code>	Borra el fichero o directorio
<code>boolean isFile()</code>	Investiga si es un fichero
<code>boolean isDirectory()</code>	Investiga si es un directorio
<code>long length()</code>	Tamaño del fichero en bytes
<code>long lastModified()</code>	Fecha de la última actualización
<code>boolean renameTo()</code>	Cambia el nombre del fichero

6. Gestión de entrada/salida

Entrada/salida estándar

- Receta #4. Escritura por la salida estándar y de error:

```
System.out.println("Using standard output ending with carriage return");  
System.out.print("Using standard output ending without carriage return");  
System.err.println("Using standard error ending with carriage return");  
System.err.print("Using standard error ending without carriage return");
```

- Receta #5. Lectura por la entrada estándar mediante la clase `java.util.Scanner`:

```
Scanner sc = new Scanner(System.in);  
String string = sc.nextLine(); // Read one string  
int number = sc.nextInt(); // Read one integer  
sc.close();
```

Las instancias de la clase `Scanner` hay que cerrarlas (método `close()`)

6. Gestión de entrada/salida

Java NIO

- La clase `java.nio.Path` es la versión actualizada de `java.io.File`
- Los objetos de tipo `Path` típicamente los creamos usando el método estático `get` de la clase `java.nio.files.Paths`

```
Path p = Paths.get("/opt/jpgTools/README.txt");  
System.out.println(p.getParent()); // /opt/jpgTools  
System.out.println(p.getRoot()); // /  
System.out.println(p.getNameCount()); // 3  
System.out.println(p.getName(0)); // opt  
System.out.println(p.getName(1)); // jpgTools  
System.out.println(p.getFileName()); // README.txt  
System.out.println(p.toString()); // The full path
```

6. Gestión de entrada/salida

Java NIO

- Receta #6. Lectura de un fichero de texto con NIO:

```
try {  
    Path path = Paths.get("input.txt");  
    List<String> lines = Files.readAllLines(path,  
        Charset.defaultCharset());  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

- Receta #7. Lectura de un fichero binario con NIO:

```
try {  
    Path path = Paths.get("image.png");  
    byte[] bytes = Files.readAllBytes(path);  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

6. Gestión de entrada/salida

Java NIO

- **Receta #8. Escritura de un fichero de texto con NIO:**

```
try {  
    List<String> list = new ArrayList<String>();  
    list.add("line1");  
    list.add("line2");  
    Files.write(Paths.get("output-nio.txt"), list,  
                Charset.defaultCharset());  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

- **Receta #9. Escritura de un fichero binario con NIO:**

```
try {  
    byte[] bytes = { 0, 1, 2, 3, 4 };  
    Files.write(Paths.get("output-nio.bin"), bytes);  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

6. Gestión de entrada/salida

Java NIO

- Receta #10. Anexar en un fichero de texto con NIO:

```
try {  
    List<String> list = new ArrayList<String>();  
    list.add("line1");  
    list.add("line2");  
    Files.write(Paths.get("output-nio.txt"), list,  
                Charset.defaultCharset(), StandardOpenOption.APPEND);  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

- Receta #11. Creación de un fichero en un directorio temporal:

```
try {  
    Path tempFile = Files.createTempFile("prefix", ".suffix");  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

Índice de contenidos

1. Introducción a Java
2. Elementos básicos
3. Clases y objetos
4. Elementos avanzados
5. Introducción a Eclipse
6. Gestión de entrada/salida
7. Colecciones
 - Genéricos
 - Listas
 - Conjuntos
 - Colas
 - Mapas
8. Programación con sockets
9. Hilos en Java
10. Para ampliar

7. Colecciones

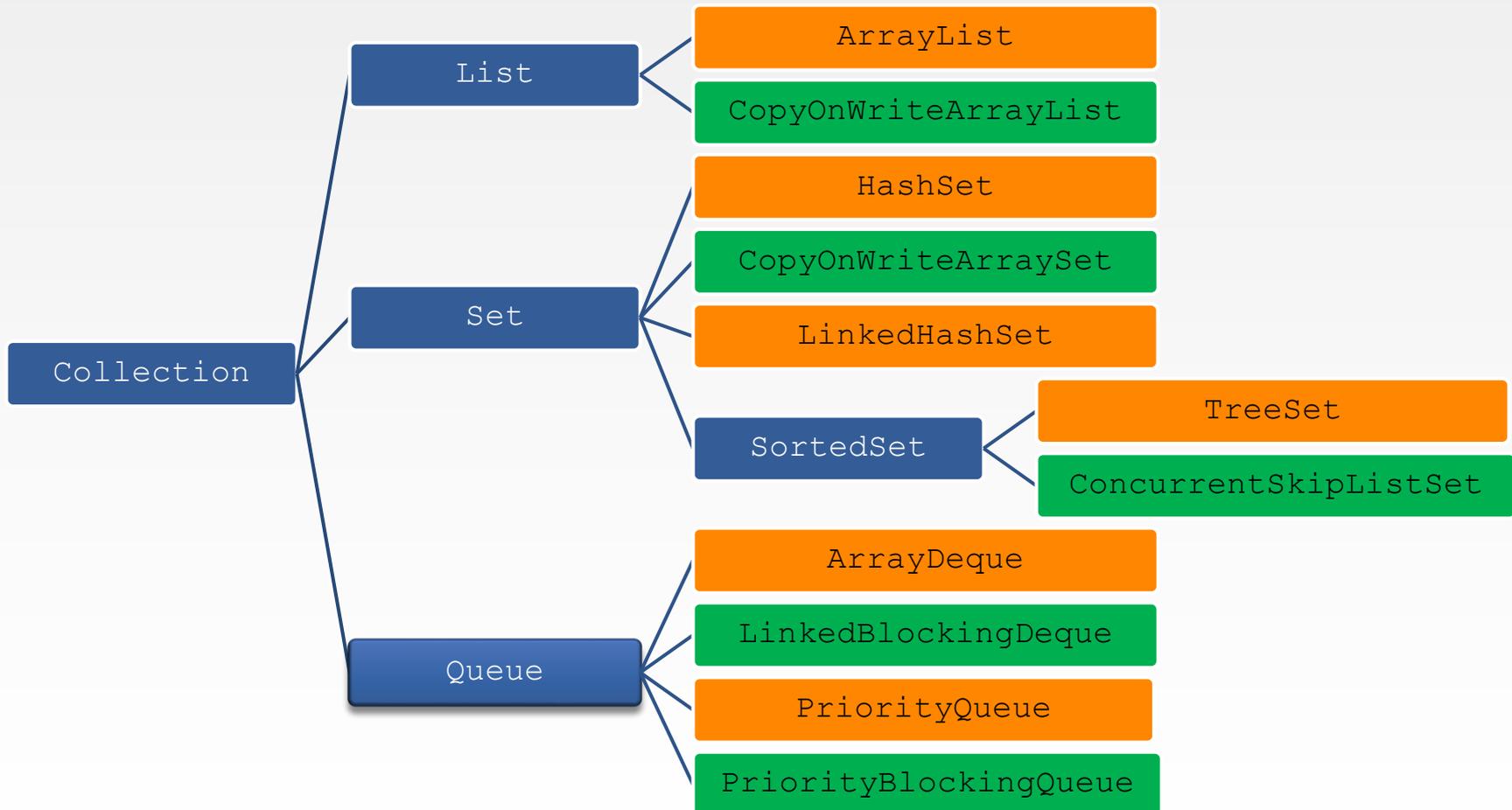
- Una colección es un grupo de elementos
- Los tipos de colecciones vienen representados por interfaces (`java.util.*`) que se clasifican en los siguientes tipos:
 - **Listas** (*lists*): Colección ordenada (secuencia). Permiten duplicados. Cada elemento puede ser accedido por su índice (número entero)
 - **Conjuntos** (*sets*): Colecciones que no admiten dos elementos iguales
 - **Mapas** (*maps*): Colecciones que asocian un valor a una clave. No puede tener dos claves iguales
 - **Colas** (*queues*): Colecciones que permiten crear colas LIFO o FIFO. Solo pueden accederse a sus objetos de el principio, final o ambos, dependiendo de la implementación
 - LIFO = *Last In, First Out* (Pila)
 - FIFO = *First In, First Out* (Cola)

7. Colecciones

 = Interfaz

 = Clase

 = Clase *thread-safe*

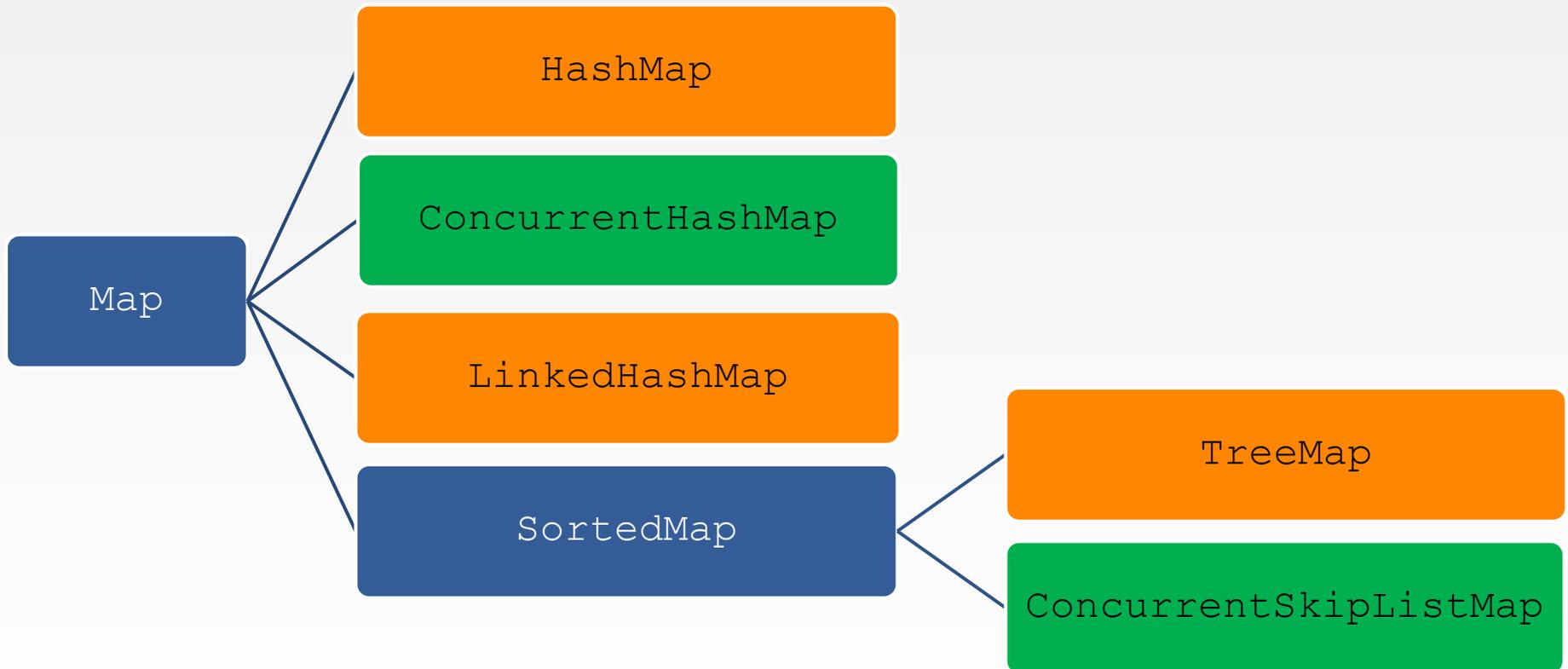


7. Colecciones

 = Interfaz

 = Clase

 = Clase *thread-safe*



7. Colecciones

Genéricos

- Los **genéricos** es un tipo de polimorfismo en Java que permite asignar parámetros a las clases, de forma que sólo admitan objetos de un determinado tipo
- Sirven para crear código más legible y robusto
- Nos vamos a encontrar los genéricos de forma mayoritaria en las colecciones
- Los genéricos se declaran usando los signos < >:

```
List<String> list = new ArrayList<>();  
Map<Integer, String> map = new HashMap<>();  
Set<String> set = new HashSet<>();  
Queue<String> queue = new ArrayDeque<>();
```

7. Colecciones

Listas

- Una lista es una colecciones que puede tener elementos duplicados
 - `ArrayList`:
 - Lista implementada con un array que se redimensiona dinámicamente
 - `CopyOnWriteArrayList`:
 - Equivalente a `ArrayList` pero sincronizado (permite acceso concurrente)

7. Colecciones

Listas

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListExample {
    public static void main(String[] args) {
        // Creation
        List<String> list = new ArrayList<>();
        // Insert
        list.add("Element 1");
        list.add("Element 2");
        list.add("Element 3");
        // Update
        list.set(1, "Element 2 updated");
        // Remove
        list.remove(2);
        // Read
        for (String s : list) {
            System.out.println(s);
        }
    }
}
```

Ejemplo de uso
de ArrayList

Salida por pantalla:

Element 1
Element 2 updated

7. Colecciones

Conjuntos

- Un conjunto es una colecciones que no tiene elementos duplicados
 - `HashSet`:
 - Implementación genérica de un conjunto. No garantiza el orden
 - `CopyOnWriteArraySet`:
 - Equivalente a `HashSet` pero sincronizado (permite acceso concurrente)
 - `LinkedHashSet`:
 - Garantiza el orden según la inserción
 - `TreeSet`:
 - Garantiza el orden ascendente de los elementos
 - `ConcurrentSkipListSet`:
 - Equivalente a `TreeSet` pero sincronizado (permite acceso concurrente)

7. Colecciones

Conjuntos

```
import java.util.HashSet;
import java.util.Set;

public class HashSetExample {

    public static void main(String[] args) {
        Set<String> set = new HashSet<String>();
        // Insert
        set.add("Element 1");
        set.add("Element 2");
        set.add("Element 3");

        // Remove
        set.remove("Element 3");

        // Read
        for (String s : set) {
            System.out.println(s);
        }
    }
}
```

Ejemplo de uso
de HashSet

Salida por pantalla:



```
Element 2
Element 1
```

7. Colecciones

Colas

- Hay dos tipos de colas: queues (proporcionan acceso al último elemento de la cola) y deque (proporcionan acceso a los dos extremos de la cola)
 - `ArrayDeque`:
 - Implementa tanto cola LIFO como FIFO
 - `LinkedBlockingDeque`:
 - Equivalente a `ArrayDeque` pero sincronizado (permite acceso concurrente)
 - `PriorityQueue`:
 - Cola donde el primer elemento no dependerá de su tiempo de inserción, sino de cualquier otro factor (tamaño, prioridad, etc.)
 - `PriorityBlockingQueue`:
 - Equivalente a `PriorityQueue` pero sincronizado (permite acceso concurrente)

7. Colecciones

Colas

Ejemplo de uso de Deque

Salida por pantalla:

```
3 one
2 two
1 three
0 ----
3 three
2 two
1 one
0 ----
```



```
import java.util.ArrayDeque;
import java.util.Collections;
import java.util.Queue;

public class ArrayDequeExample {
    public static void main(String[] args) {
        Queue<String> queue = new ArrayDeque<String>();
        queue.offer("one");
        queue.offer("two");
        queue.offer("three");
        System.out.println(queue.size() + " " + queue.poll());
        System.out.println(queue.size() + " " + queue.poll());
        System.out.println(queue.size() + " " + queue.poll());
        System.out.println(queue.size() + " ----");

        Queue<String> stack = Collections.asLifoQueue(new ArrayDeque<String>());
        stack.offer("one");
        stack.offer("two");
        stack.offer("three");
        System.out.println(stack.size() + " " + stack.poll());
        System.out.println(stack.size() + " " + stack.poll());
        System.out.println(stack.size() + " " + stack.poll());
        System.out.println(stack.size() + " ----");
    }
}
```

7. Colecciones

Mapas

- Un mapa es una colección clave-valor
 - `HashMap`:
 - Implementación genérica de un mapa. No garantiza el orden según su inserción
 - `ConcurrentHashMap`:
 - Implementación sincronizada de `HashMap` (permite acceso concurrente)
 - `LinkedHashMap`:
 - Implementación de mapa que garantiza orden de claves por su tiempo de inserción
 - `TreeMap`:
 - Implementación de mapa que garantiza el orden natural de las claves
 - `ConcurrentSkipListMap`:
 - Implementación sincronizada de `TreeMap` (permite acceso concurrente)

7. Colecciones

Mapas

```
import java.util.HashMap;
import java.util.Map;

public class HashMapExample {
    public static void main(String[] args) {
        // Creation
        Map<Integer, String> map = new HashMap<>();
        // Insert
        map.put(0, "value 0");
        map.put(1, "value 1");
        map.put(2, "value 1");
        // Update
        map.put(1, "value 1 updated");
        // Remove
        map.remove(2);
        // Read
        for (Integer i : map.keySet()) {
            System.out.println("key " + i + " - " + map.get(i));
        }
    }
}
```

Ejemplo de uso
de HashMap

Salida por pantalla:

key 0 - value 0
key 1 - value 1 updated

Índice de contenidos

1. Introducción a Java
2. Elementos básicos
3. Clases y objetos
4. Elementos avanzados
5. Introducción a Eclipse
6. Gestión de entrada/salida
7. Colecciones
- 8. Programación con sockets**
 - La API de sockets
 - La clase `InetAddress`
 - Sockets TCP
 - Sockets UDP
9. Hilos en Java
10. Para ampliar

8. Programación con sockets

La API de sockets

- Un **socket** es una abstracción que proporciona un mecanismo para la comunicación entre procesos distribuidos
- Los sockets aparecen en Unix BSD 4.2 al integrar TCP/IP en el sistema operativo
- Para establecer una comunicación por sockets es necesario saber:
 - Protocolo de transporte (TCP, UDP)
 - Dirección IP local
 - Puerto local
 - Dirección IP remota
 - Puerto remoto

8. Programación con sockets

La API de sockets

- La **API de sockets** es una interfaz de acceso al servicio que ofrecen los protocolos de nivel de transporte



8. Programación con sockets

La API de sockets

- Para desarrollar una aplicación que usa la API de sockets hay que crear un par de programas (cliente y servidor)
- Cuando se ejecutan se crean dos procesos (cliente y servidor)
- Estos dos procesos se comunican entre sí leyendo y escribiendo en su socket
- Cuando un programa cliente y un programa servidor implementan el mismo protocolo previamente definido, se pueden comunicar
- El puerto a utilizar debe estar definido en el protocolo

8. Programación con sockets

La API de sockets

- **Paquete** `java.net`:
 - `InetAddress` : Se encarga de implementar la dirección IP
 - `Socket` : Implementa un extremo cliente de la conexión TCP
 - `ServerSocket` : Implementa el extremo servidor de la conexión TCP
 - `DatagramSocket` : Implementa tanto el servidor como el cliente cuando se utiliza UDP
 - `DatagramPacket` : Implementa un datagrama, que se utiliza para el intercambio de datos en UDP

8. Programación con sockets

La clase `InetAddress`

- Crea objetos dirección de red
- Soporta direcciones IP y nombres de dominio. Por ejemplo:
 - `123.152.4.5`
 - `maquina.dominio.es`
- Soporta direcciones IPv4 e IPv6
- No tiene constructores públicos. Sólo pueden usarse sus métodos estáticos para crear instancias

8. Programación con sockets

La clase InetAddress

Retorno	Método	Función
Static InetAddress	<code>getByName(String host)</code>	Determina la dirección IP dado el nombre del host
Static InetAddress[]	<code>getAllByName(String host)</code>	Determina todas las direcciones IP dado el nombre del host
Static InetAddress	<code>getByAddress(byte[] address)</code>	Crea un objeto con la dirección dada
String	<code>getHostName()</code>	Devuelve el nombre del host
byte[]	<code>getAddress()</code>	Devuelve la dirección IP
String	<code>getHostAddress()</code>	Devuelve la dirección IP como un String

8. Programación con sockets

La clase InetAddress

```
import java.net.InetAddress;
import java.net.UnknownHostException;

public class DnsQuery01 {

    public static void main(String[] args) throws UnknownHostException {
        InetAddress address = InetAddress.getByName("www.google.es");
        System.out.println(address.getHostAddress());
    }
}
```

Salida por pantalla:

216.58.210.227

8. Programación con sockets

La clase InetAddress

```
import java.net.InetAddress;
import java.net.UnknownHostException;

public class DnsQuery02 {

    public static void main(String[] args) throws UnknownHostException {
        InetAddress[] addresses = InetAddress.getAllByName("amazon.com");
        for (InetAddress i : addresses) {
            System.out.println(i.getHostAddress());
        }
    }
}
```

Salida por pantalla:

```
205.251.242.54
176.32.98.166
176.32.103.205
```

8. Programación con sockets

Sockets TCP

- El API Java para TCP proporciona la abstracción de un flujo de bytes (*stream*) en el que pueden escribirse y desde el que pueden leerse datos
- Un par de procesos establecen una conexión antes de empezar a comunicarse
- El API para la comunicación por streams necesita disponer de un cliente y un servidor para establecer la conexión, aunque después se comuniquen de igual a igual
- Cada conector tiene su propio stream de entrada y de salida
- Uno de los procesos manda información al otro escribiendo en su stream de salida
- El otro proceso obtiene la información leyendo de su stream de entrada

8. Programación con sockets

Sockets TCP

- Para identificar al destino de los paquetes de datos utilizan dirección IP y puerto
- Sockets distintos en el mismo puerto tienen colas de paquetes independientes
- El servidor ofrece un servicio al cliente
 - El servidor debe tener una dirección conocida por el cliente (dirección IP o nombre de dominio)
- Debe estar ejecutándose antes que los clientes
- Varios clientes pueden enviar datos al mismo proceso servidor (a la misma aplicación)

8. Programación con sockets

Sockets TCP

- El API Java para sockets TCP está constituida por dos clases:
- `ServerSocket`
 - Utilizada como servidor, escucha un puerto en el lado servidor
 - Maneja una cola de conexiones entrantes
 - Si la cola está vacía se bloquea hasta que llegue una petición
 - Su método `accept` toma una petición de esta cola. El resultado de este método es una instancia de `Socket`, mediante la que se puede establecer la comunicación con el cliente
- `Socket`
 - Utilizada tanto en el cliente como en el servidor
 - Proporciona los métodos `getInputStream` y `getOutputStream` para usar los flujos de bytes

8. Programación con sockets

Sockets TCP

- Clase `ServerSocket`

- Constructor:

```
// crea un socket servidor en la máquina local en el puerto especificado  
public ServerSocket(int port) throws IOException;
```

- Métodos importantes:

```
// aceptar conexión  
public Socket accept() throws IOException;  
  
// cierra el socket  
public void close() throws IOException;
```

8. Programación con sockets

Sockets TCP

- Clase Socket

- Constructores:

```
// Crea una conexión con la máquina local con una máquina remota
public Socket(String host, int port) throws UnknownHostException, IOException;
public Socket(InetAddress address, int port) throws IOException;
```

- Método para cerrar un socket:

```
// Cierra el socket
public synchronized void close() throws IOException;
```

- Métodos para obtener flujos de entrada/salida:

```
// Devuelve un stream de entrada (de lectura)
public InputStream getInputStream() throws IOException;

// Devuelve un stream de salida (de escritura)
public OutputStream getOutputStream() throws IOException;
```

8. Programación con sockets

Sockets TCP

- Clase Socket
 - Métodos para leer de las características del socket:

```
// Obtiene dirección destino
public InetAddress getInetAddress();

// Obtiene dirección local
public InetAddress getLocalAddress();

// Obtiene puerto destino
public int getPort();

// Obtiene puerto local
public int getLocalPort();

// Está cerrado?
public boolean isClosed();

// Está conectado?
public boolean isConnected();
```

8. Programación con sockets

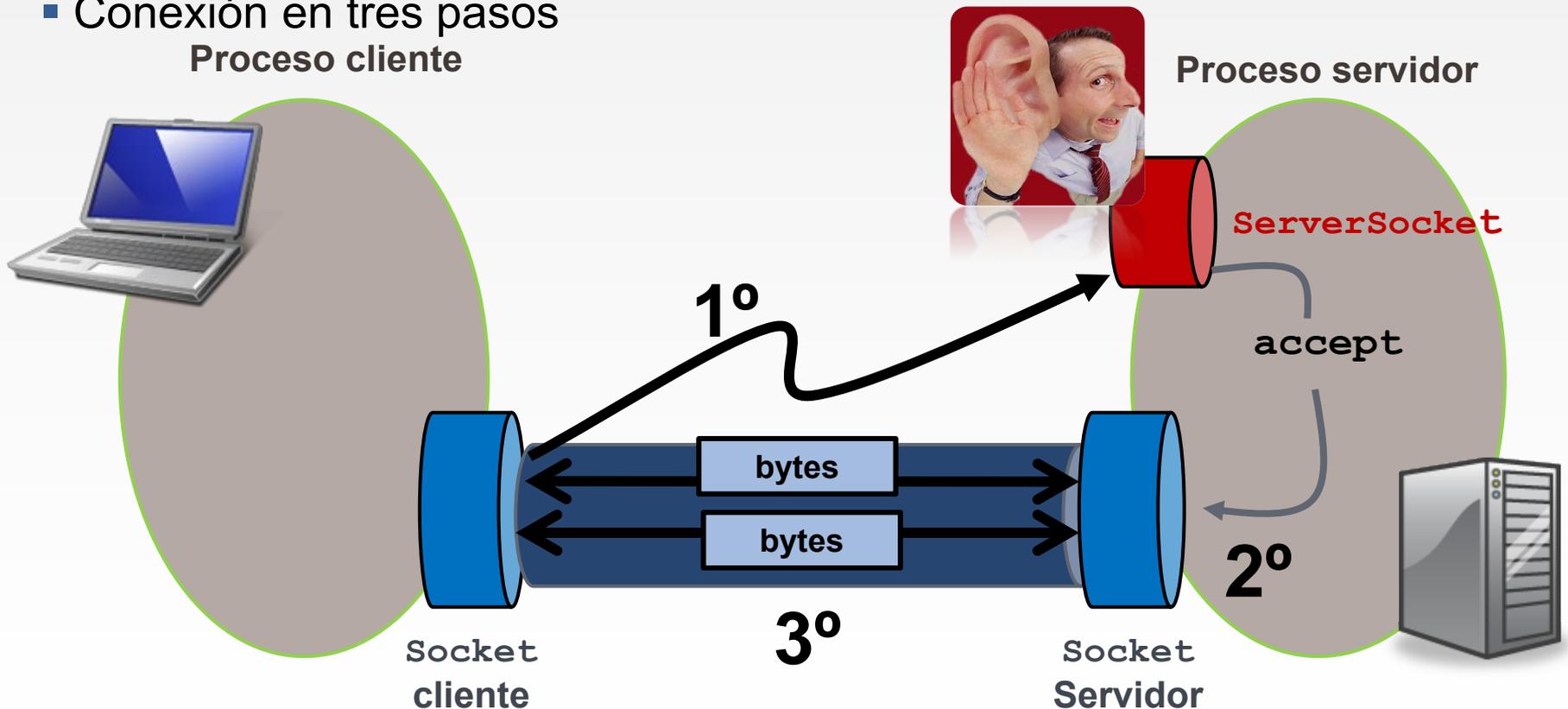
Sockets TCP

- Implementación del servidor
 1. Creación del socket del servidor (`ServerSocket`)
 2. Espera solicitud de conexión entrante (la llegada de un cliente)
 3. Si llega conexión, la acepta (`accept`)
 - Se genera un socket con la conexión establecida (punto de conexión con el cliente en el servidor)
 - Obtenemos los flujos de entrada y salida a partir del socket para el intercambio de datos
 4. Comunicación entre cliente-servidor
 5. Cuando acaba → Liberación de la conexión

8. Programación con sockets

Sockets TCP

- Conexión en tres pasos
Proceso cliente



8. Programación con sockets

Sockets TCP

- Clase `ServerSocket`

- Implementación del servidor

1. Creación del socket del servidor (`ServerSocket`)

```
ServerSocket server = new ServerSocket(12345);
```

2. Espera solicitud de conexión entrante (la llegada de un cliente)

```
Socket socket = server.accept();
```

- Permanece bloqueado hasta que se conecta un cliente
- Cuando se conecta un cliente y “lo acepta” devuelve un socket de conexión con este cliente
- Si quisiésemos aceptar varios clientes simultáneamente tendríamos que utilizar hilos (lo veremos en la sección 9)

En este ejemplo el servidor escucha en el puerto 12345

8. Programación con sockets

Sockets TCP

- Clase `ServerSocket`
 - Implementación del servidor
 - 3. Obtener flujos de datos (leer/escribir al cliente):

(para leer/escribir cadenas de caracteres)

```
// Flujo de escritura
PrintWriter output = new PrintWriter(socket.getOutputStream(), true);
String message = input.readLine();

// Flujo de lectura
BufferedReader input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
output.println(message);
```

Utilizamos diferentes clases filtros de I/O según el tipo de flujo de datos que se transmita/reciba

8. Programación con sockets

Sockets TCP

- Clase `ServerSocket`
 - Implementación del servidor
 - 3. Obtener flujos de datos (leer/escribir al cliente):

(para leer/escribir tipos primitivos)

```
// Flujo de escritura
DataOutputStream output = new DataOutputStream(socket.getOutputStream());
output.writeInt(i);
output.writeFloat(f);

// Flujo de lectura
DataInputStream input = new DataInputStream(socket.getInputStream());
float f = input.readFloat();
int i = input.readInt();
```

8. Programación con sockets

Sockets TCP

- Clase `ServerSocket`
 - Implementación del servidor
 - 3. Obtener flujos de datos (leer/escribir al cliente):

(para leer/escribir objetos)

```
// Flujo de escrituras
ObjectOutputStream output = new ObjectOutputStream(socket.getOutputStream());
MyClass data = new MyClass();
output.writeObject(data);

// Flujo de lectura
ObjectInputStream input = new ObjectInputStream(socket.getInputStream());
MyClass data = (MyClass) input.readObject();
```

Para leer el objeto de tipo esperado hay que realizar una conversión de tipos (**casting**)

8. Programación con sockets

Sockets TCP

- Ejemplo servidor
- Servicio de “echo”

Uso especial del bloque `try` (*try-with-resources*) en el que se define un recurso que se cierra (*closeable*) al final del bloque

Servidor en bucle infinito (recibiendo peticiones indefinidamente)

```
import java.io.*;
import java.net.*;

public class EchoServer {
    public static void main(String[] args) throws IOException {
        try (ServerSocket server = new ServerSocket(12345)) {
            while (true) {
                System.out.println("Waiting for incoming messages...");
                Socket socket = server.accept();
                PrintWriter output = new PrintWriter(socket.getOutputStream(),
                    true);
                BufferedReader input = new BufferedReader(
                    new InputStreamReader(socket.getInputStream()));
                String message = input.readLine();
                System.out.println("Message received " + message);
                System.out.println("Sending back message " + message);
                output.println(message);
                input.close();
                output.close();
                socket.close();
            }
        }
    }
}
```

8. Programación con sockets

Sockets TCP

- Ejemplo cliente
- Cliente de “echo”

```
import java.io.*;
import java.net.*;

public class EchoClient {

    public static void main(String[] args) throws IOException {
        Socket client = new Socket("localhost", 12345);
        PrintWriter output = new PrintWriter(client.getOutputStream(), true);
        BufferedReader input = new BufferedReader(
            new InputStreamReader(client.getInputStream()));
        String message = "Hello server!";
        System.out.println("Sending message to server: " + message);
        output.println(message);
        String returnedMessage = input.readLine();
        System.out.println("Returned message: " + returnedMessage);
        output.close();
        input.close();
        client.close();
    }
}
```

8. Programación con sockets

Sockets TCP

- Ejecución de cliente-servidor de echo:

1. Ejecutamos servidor:

```
Waiting for incoming messages...
```

2. Ejecutamos cliente (el proceso termina):

```
Sending message to server: Hello server!  
Returned message: Hello server!
```

3. Llega petición a servidor (el proceso continúa):

```
Waiting for incoming messages...  
Message received Hello server!  
Sending back message Hello server!
```

8. Programación con sockets

Sockets TCP

- Otro ejemplo
- Servidor de echo de “personas”

```
import java.io.*;
import java.net.*;
public class PeopleEchoServer {
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        ServerSocket server = new ServerSocket(12345);
        System.out.println("Waiting for incoming people...");
        while (true) {
            Socket socket = server.accept();
            System.out.println("Request received");
            // Flujo de escritura
            ObjectOutputStream output = new ObjectOutputStream(socket.getOutputStream());
            // Flujo de lectura
            ObjectInputStream input = new ObjectInputStream(socket.getInputStream());
            Person person = (Person) input.readObject();
            System.out.println("Person received " + person);
            System.out.println("Sending back person " + person);
            output.writeObject(person);
            input.close();
            output.close();
            socket.close();
        }
    }
}
```

8. Programación con sockets

Sockets TCP

- Cliente de echo de “personas”

```
import java.io.*;
import java.net.*;
public class PeopleEchoClient {
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        Socket socket = new Socket("localhost", 12345);
        // Flujo de escritura
        ObjectOutputStream output = new ObjectOutputStream(
            socket.getOutputStream());
        // Flujo de lectura
        ObjectInputStream input = new ObjectInputStream(socket.getInputStream());
        Person person = new Person("John", "Smith");
        System.out.println("Sending person to server: " + person);
        output.writeObject(person);
        Person returned = (Person) input.readObject();
        if (returned != null) {
            System.out.println("Returned person: " + returned);
        }
        output.close();
        input.close();
        socket.close();
    }
}
```

8. Programación con sockets

Sockets TCP

- Clase serializable

Una clase que implementa el interfaz `Serializable` permite transformar un objeto a una secuencia de bytes que puede ser posteriormente leído para obtener el objeto original

```
import java.io.Serializable;
public class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private String surname;
    public Person(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getSurname() {
        return surname;
    }
    public void setSurname(String surname) {
        this.surname = surname;
    }
    @Override
    public String toString() {
        return this.getName() + " " + this.getSurname();
    }
}
```

Se recomienda definir el atributo `serialVersionUID`, que sirve para asegurar compatibilidad entre las versiones de las clases implicadas

8. Programación con sockets

Sockets UDP

- Recordatorio: UDP es un protocolo **no orientado a la conexión**
 - No requiere conexión previa
 - No ofrece un servicio fiable de entrega
 - Cada datagrama UDP es independiente de los demás
- Los sockets UDP en Java se gestionan mediante dos clases:
 - DatagramSocket: Realiza la comunicación
 - DatagramPacket: Contiene el mensaje y dirección destino



8. Programación con sockets

Sockets UDP

- **Clase** `DatagramSocket`
 - Sirven para enviar y recibir datagramas UDP
 - Tanto cliente como servidor usan esta clase para enviar/recibir
- **Clase** `DatagramPacket`
 - Sirve para modelar el mensaje que transporta un socket UDP
 - Se obtiene mediante el método `getData` de `DatagramSocket`
 - Los métodos `getPort` y `getAddress` acceden respectivamente al puerto y a la dirección
 - Se instancia de distinta forma si es para enviar o recibir

8. Programación con sockets

Sockets UDP

- Clase DatagramSocket

- Constructores:

```
// Crea un socket en algún puerto libre de la máquina local (localhost=127.0.0.1)
public DatagramSocket() throws SocketException;

// Crea el socket en la máquina local el puerto que se le pasa como parámetro
// Si se asigna el puerto 0 o se invoca al constructor sin argumentos
// (cualquier puerto libre)
public DatagramSocket(int port) throws SocketException;
```

- Métodos:

```
public void send(DatagramPacket p) throws IOException; // Envía un datagrama
public void receive(DatagramPacket p) throws IOException; // Recibe un datagrama
public void close(); // Cierra socket
```

8. Programación con sockets

Sockets UDP

- **Clase** DatagramPacket

- **Constructores:**

```
// Constructores para recepción
public DatagramPacket(byte data[], int length);

// Constructores para envío
public DatagramPacket(byte data[], int length, InetAddress destination, int port);
```

- **Métodos:**

```
public InetAddress getAddress(); // Obtiene dirección de red
public int getPort(); // Obtiene puerto
public byte[] getData(); // Recibir datos
public void setAddress(InetAddress addr); // Dirección destino
public void setPort(int port); // Puerto destino
public void setData(byte[] data); // Datos a enviar
```

8. Programación con sockets

Sockets UDP

▪ Envío de datagramas

1. Se crea un socket UDP (`DatagramSocket`)
2. Se crea un datagrama (`DatagramPacket`). Hay que especificar:
 - El mensaje a enviar (array de bytes)
 - La dirección de red y puerto al que transmitir
3. Se envía el datagrama (`send`) por el socket

▪ Recepción de datagramas

1. Se crea un socket UDP (`DatagramSocket`)
2. Se crea un datagrama (`DatagramPacket`). Hay que especificar:
 - Un buffer de recepción vacío (de tamaño mayor al tamaño del datagrama esperado)
3. Se espera la recepción de un datagrama (`receive`) en el socket

8. Programación con sockets

Sockets UDP

- Recepción de datagramas (servidor)

1. Se crea un socket UDP (DatagramSocket):

```
DatagramSocket socket = new DatagramSocket(12346);
```

2. Se crea un datagrama (DatagramPacket):

```
byte[] buffer = ...  
DatagramPacket rcvDatagram = new DatagramPacket(buffer, buffer.length);
```

3. Se espera la recepción de un datagrama (receive) en el socket:

```
socket.receive(rcvDatagram);
```

→Esta sentencia bloquea el programa hasta que se reciben datos

8. Programación con sockets

Sockets UDP

- Envío de datagramas (cliente)

1. Se crea un socket UDP (DatagramSocket):

```
DatagramSocket socket = new DatagramSocket();
```

2. Se crea un datagrama (DatagramPacket):

```
byte[] sendBuffer = ...  
DatagramPacket sendDatagram = new DatagramPacket(sendBuffer,  
    sendBuffer.length, serverAddress, serverPort);
```

3. Se envía el datagrama (send) por el socket:

```
socket.send(sendDatagram);
```

8. Programación con sockets

Sockets UDP

- Los mensajes a enviar van contenidos en arrays de bytes (`byte[]`)
- Todo lo que recibamos, al hacer `.getData()` del `DatagramPacket` recibido, lo obtenemos en un array de bytes
- Envío/recepción de `String`
 - Para enviar un `String` lo convierto a bytes:

```
String message = "hello";  
byte[] buff = message.getBytes();
```

- Si los bytes que recibo los quiero convertir a `String`:

```
String strDatagram = new String(rcvDatagram.getData());
```

8. Programación con sockets

Sockets UDP

- Envío/recepción de tipo primitivos:
 - Para enviar, usamos las clases `ByteArrayOutputStream` y `DataOutputStream` para convertir tipos primitivos a array de bytes:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream daos = new DataOutputStream(baos);
daos.writeInt(1);
daos.writeBoolean(true);
// ...
byte[] buffer = baos.toByteArray();
```

- Para recibir, usamos las clases `ByteArrayInputStream` y `DataInputStream`:

```
ByteArrayInputStream bais = new ByteArrayInputStream(rcvDatagram.getData());
DataInputStream dis = new DataInputStream(bais);
float num = dis.readFloat();
```

8. Programación con sockets

Sockets UDP

- Envío de objetos (serializables):
 - Para enviar lo convertimos a array de bytes:

```
Person p = new Person();  
byte[] buff = d.toByteArray();
```

```
public byte[] toByteArray(Person person) throws IOException {  
    try (ByteArrayOutputStream baos = new ByteArrayOutputStream();  
         ObjectOutputStream out = new ObjectOutputStream(baos)) {  
        out.writeObject(person);  
        return baos.toByteArray();  
    }  
}
```

Añadimos el método para convertir el objeto a array de bytes en la clase serializable

8. Programación con sockets

Sockets UDP

- **Recepción de objetos (serializables):**
 - Para recibir, usamos las clases `ByteArrayInputStream` y `ObjectInputStream`:

```
ByteArrayInputStream byteArray = new ByteArrayInputStream(rcvDatagram.getData());  
ObjectInputStream is = new ObjectInputStream(byteArray);  
Person person = (Person) is.readObject();
```

8. Programación con sockets

Sockets UDP

- Ejemplo: servidor
- Ofrece un servicio de fecha a sus clientes

```
public class UdpDateServer {
    public static void main(String[] args) throws IOException {
        // Server socket creation
        try (DatagramSocket socket = new DatagramSocket(12346)) {
            // Receive datagram
            byte[] buffer = new byte[256];
            DatagramPacket rcvDatagram = new DatagramPacket(buffer,
                buffer.length);
            System.out.println("UDP Date Server ready");
            while (true) {
                // Waiting to receive incoming datagrams
                socket.receive(rcvDatagram);
                // Received datagram
                String strDatagram = new String(rcvDatagram.getData());
                System.out.println("Received " + strDatagram + " from "
                    + rcvDatagram.getAddress() + ":"
                    + rcvDatagram.getPort());
                // Sending back the current date
                buffer = (new Date()).toString().getBytes();
                socket.send(new DatagramPacket(buffer, buffer.length,
                    rcvDatagram.getAddress(), rcvDatagram.getPort()));
            }
        }
    }
}
```

8. Programación con sockets

Sockets UDP

- Ejemplo: cliente
- Consume un servicio de fecha

```
public class UdpDateClient {  
  
    public static void main(String[] args) throws IOException {  
        // Client socket (localhost, any available port)  
        DatagramSocket socket = new DatagramSocket();  
        // Server address and port  
        InetAddress serverAddress = InetAddress.getByName("localhost");  
        int serverPort = 12346;  
        // Send request to date server  
        String message = "What date is it?";  
        byte[] sendBuffer = message.getBytes();  
        DatagramPacket sendDatagram = new DatagramPacket(sendBuffer,  
            sendBuffer.length, serverAddress, serverPort);  
        socket.send(sendDatagram);  
        // Wait for response  
        byte[] rcvBuffer = new byte[1024];  
        DatagramPacket rcvDatagram = new DatagramPacket(rcvBuffer,  
            rcvBuffer.length);  
        socket.receive(rcvDatagram);  
        String strDatagram = new String(rcvDatagram.getData());  
        System.out.println("Response from server: " + strDatagram);  
        socket.close();  
    }  
}
```

Índice de contenidos

1. Introducción a Java
2. Elementos básicos
3. Clases y objetos
4. Elementos avanzados
5. Introducción a Eclipse
6. Gestión de entrada/salida
7. Colecciones
8. Programación con sockets
9. Hilos en Java
 - Programación multihilo
 - Procesos
 - Hilos
 - Creación de hilos
 - ¿Para qué utilizar hilos?
 - Ejemplo de servidor concurrente
10. Para ampliar

9. Hilos en Java

Programación multihilo

- Normalmente los programas tienen un comportamiento secuencial y un único “camino” de ejecución, tienen:
 - Un punto de inicio
 - Una secuencia de ejecución
 - Un final
- Cuando en un programa se quiere realizar más de una tarea simultáneamente se utilizan técnicas de ejecución que permiten que diversos puntos del programa se estén ejecutando a la vez
- La programa multihilo contiene dos o mas “partes” que pueden ejecutarse de forma simultánea. Permite realizar muchas actividades de forma simultánea en un programa
- Java proporciona soporte para la programación multihilo

9. Hilos en Java

Procesos

- Los procesos son programa en ejecución
- Recursos propios que se crean durante la vida de un proceso y se destruyen cuando finaliza:
 - Espacio propio de direcciones
 - Memoria
 - Variables
 - Ficheros
 - Registros, señales, etc.
- En sistemas operativos, un proceso es la unidad mínima de código que puede seleccionar el planificador, cuando hablamos de un sistema multitarea.
- La multitarea basada en procesos no está bajo control de Java

9. Hilos en Java

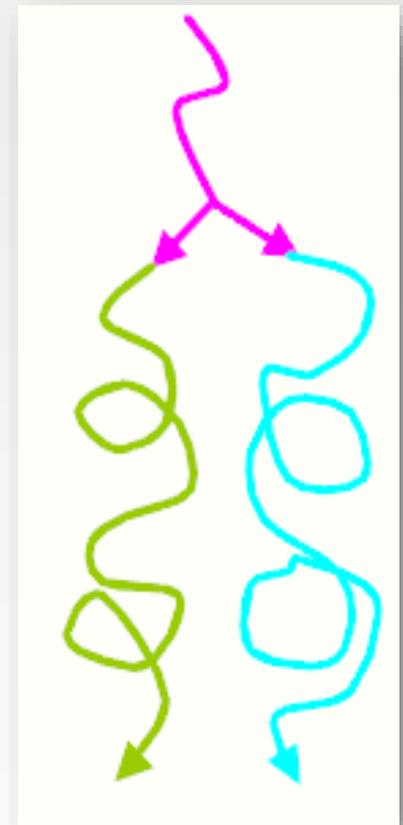
Hilos

- Unidades concurrentes dentro de un programa. Un único programa puede realizar dos o más tareas de forma simultánea (hilos)
- Son tareas ligeras: bajo coste en consumo de recursos, que permiten el uso óptimo de la CPU
- Comparten recursos con otros hilos y con el proceso:
 - Mismo espacio de direcciones
 - Mismo espacio de memoria
 - Mismas variables globales
 - Mismo conjunto de ficheros abiertos
 - Señales, etc.
- Al compartir recursos, los hilos se crean más rápido que los procesos y los cambios de contexto entre hilos son menos costosos

9. Hilos en Java

Hilos

- Un programa que ejecuta paralelamente distintos hilos, se divide en algún momento en varios hilos (*threads*) que se ejecutan simultáneamente
- ¿Para qué?
 - Programas que tengan que realizar varias tareas de forma simultánea.
 - Servidores concurrentes
- Problemas: Acceso simultáneo a recursos compartidos
 - Necesidad de bloqueo de uno de ellos durante unos segundos, hasta que la operación del otro se haya actualizado.



9. Hilos en Java

Creación de hilos

- Formas de creación de un hilo:
 - Mediante herencia extendiendo de la clase `java.lang.Thread`
 - Implementando la interfaz `java.lang.Runnable`
- Algunos métodos de la clase `Thread`:
 - `getName`: Obtiene el nombre de un hilo
 - `isAlive`: Determina si un hilo todavía se está ejecutando
 - `join`: Espera la terminación de un hilo
 - `sleep`: suspende un hilo durante un tiempo determinado
 - `start`: comienza un hilo, llamando a su método `run`

9. Hilos en Java

Creación de hilos

- Un hilo de ejecución es un objeto de la clase `java.lang.Thread`
- Como cualquier otro objeto, se puede almacenar en una variable, tiene constructor, métodos, etc...
- El código que se ejecutará en el nuevo hilo de ejecución se indica en el constructor con un objeto de una clase que implemente el interfaz `java.lang.Runnable`
- Cuando se ejecuta el método `start()` del objeto `Thread` se crea un hilo y se ejecuta el código del método `run()` del `Runnable`

9. Hilos en Java

Creación de hilos

```
public class ThreadExample {  
    public static void main(String[] args) {  
        Thread thread = new Thread(new MyThread());  
        thread.start();  
    }  
}
```

El método `start()` del objeto `Thread` comienza la ejecución del hilo

```
public class MyThread implements Runnable {  
    @Override  
    public void run() {  
        System.out.println("I am a thread");  
    }  
}
```

La lógica del hilo se codifica en una clase que implementa el interfaz `Runnable`

El método `run()` se ejecutará cuando comience la ejecución del hilo

9. Hilos en Java

¿Para qué utilizar hilos?

- Programación sin hilos explícitos (todo en el mismo hilo)



- El servidor recibe y procesa solicitudes de los clientes
- Problemas de latencia
 - Si el servidor está ocupado procesando una solicitud no puede procesar otras
 - Reducción de disponibilidad global del servicio

9. Hilos en Java

¿Para qué utilizar hilos?

- Programación con hilos para procesar peticiones



- El hilo procesa la petición del cliente, y el programa servidor no se queda bloqueado en esa tarea, puede seguir atendiendo
- Elimina tiempos de espera a los usuarios y cuellos de botella
- Aunque el procesamiento tenga un coste temporal elevado, el servidor puede seguir admitiendo nuevas solicitudes (mejora la disponibilidad)

9. Hilos en Java

¿Para qué utilizar hilos?

- Usando hilo el servidor puede procesar peticiones simultáneas de varios clientes (servidor concurrente)



9. Hilos en Java

Ejemplo de servidor concurrente

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class ConcurrentServer {
    public static void main(String[] args) throws IOException {
        int port = 12345;
        ServerSocket serverSocket = new ServerSocket(port);
        while (true) {
            Socket socket = serverSocket.accept();
            Thread t = new Thread(new ServerThread(socket));
            t.start();
        }
    }
}
```

9. Hilos en Java

Ejemplo de servidor concurrente

```
import java.io.*;
import java.net.Socket;

public class ServerThread implements Runnable {
    private Socket socket;

    public ServerThread(Socket socket) {
        this.socket = socket;
    }

    public void run() {
        try {
            PrintWriter output = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            // Processing ...
            input.close();
            output.close();
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

9. Hilos en Java

Ejemplo de servidor concurrente

- En este ejemplo muestra como instanciar y arrancar el hilo que procesa la petición en la misma clase

Esta técnica de creación de una clase dentro de otra se conoce como **clase anónima**

Para poder usar variables definidas en la clase principal dentro de la clase anónima hay que declararlas como **final**

```
import java.io.*;
import java.net.*;

public class ConcurrentServerSingleClass {
    public static void main(String[] args) throws IOException {
        int port = 12345;
        ServerSocket serverSocket = new ServerSocket(port);
        while (true) {
            final Socket socket = serverSocket.accept();
            new Thread() {
                public void run() {
                    PrintWriter output;
                    try {
                        output = new PrintWriter(socket.getOutputStream(), true);
                        BufferedReader input = new BufferedReader(
                            new InputStreamReader(socket.getInputStream()));
                        // Processing ...
                        input.close();
                        output.close();
                        socket.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }.start();
        }
    }
}
```

Índice de contenidos

1. Introducción a Java
2. Elementos básicos
3. Clases y objetos
4. Elementos avanzados
5. Introducción a Eclipse
6. Gestión de entrada/salida
7. Colecciones
8. Programación con sockets
9. Hilos en Java
- 10. Para ampliar**

10. Para ampliar

Algunas cosas que no vemos en clase

- Concurrencia en Java
- Acceso a bases de datos (JDBC, ORM)
- Aplicaciones web con Java (servlets, ...)
- Interfaces gráficas (Swing)
- Reflexión en Java (*reflection*)
- Expresiones lambda (Java 8)
- Colecciones de tipo `Stream` (Java 8)
- Uso de `Optional` (Java 8)
- Modularidad (Java 9)
- ...