

Servicios de Internet con Java, SpringBoot & Kubernetes

Tema 1. Java Moderno

Evolución de Java

Evolución de Java

- La primera versión de Java (beta) fue publicada en **1995** (hace 24 años)
- La última versión de Java (la 13) ha sido publicada esta semana (**17 Sept de 2019**)



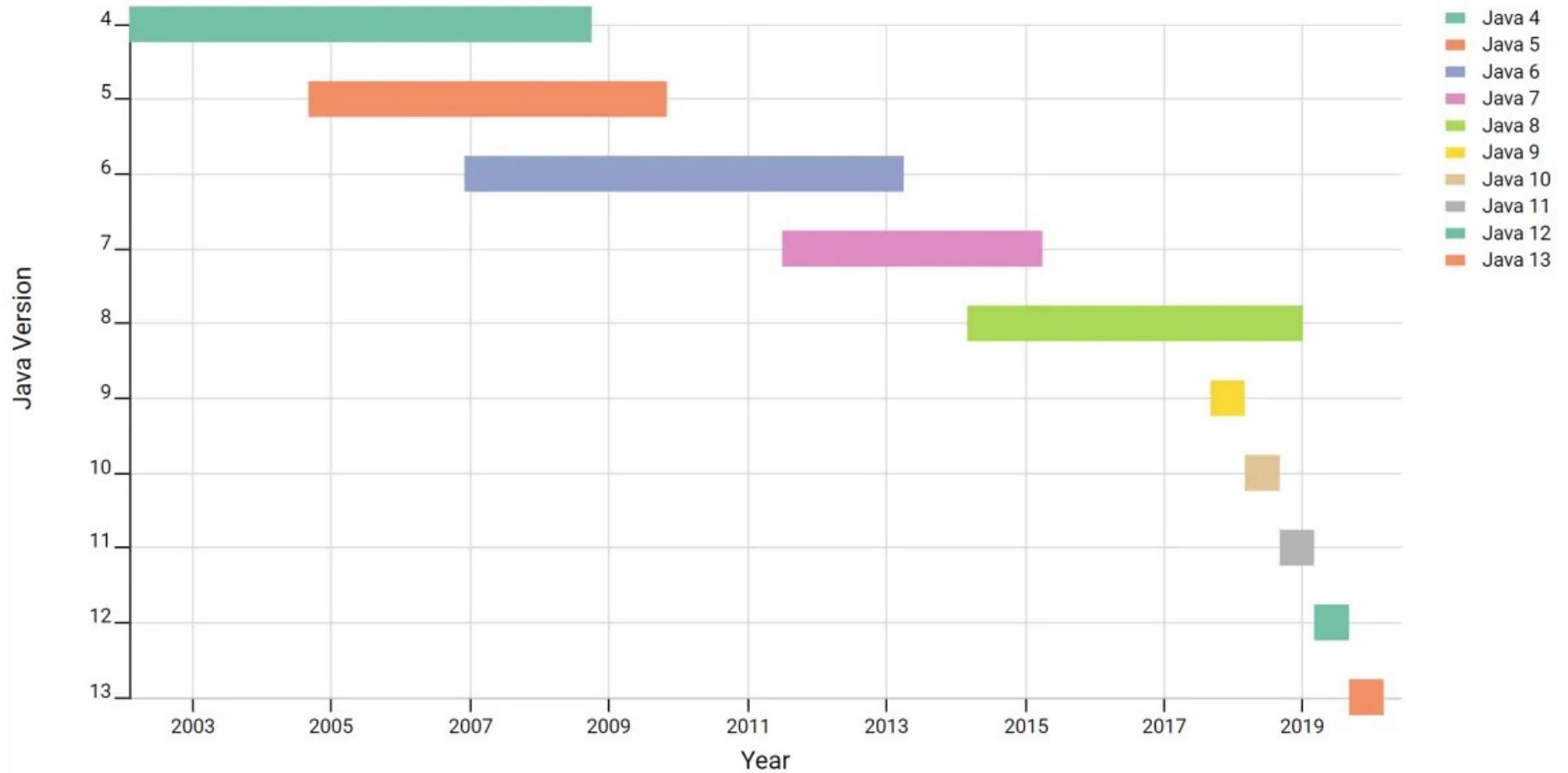
Version	Release date
JDK Beta	1995
JDK 1.0	January 1996
JDK 1.1	February 1997
J2SE 1.2	December 1998
J2SE 1.3	May 2000
J2SE 1.4	February 2002
J2SE 5.0	September 2004
Java SE 6	December 2006
Java SE 7	July 2011
Java SE 8 (LTS)	March 2014
Java SE 9	September 2017
Java SE 10	March 2018
Java SE 11 (LTS)	September 2018
Java SE 12	March 2019
Java SE 13	17 September 2019
Java SE 14 (Early-Access Builds 12) ^[9]	28 August 2019

Durante una temporada las nuevas versiones tardaban mucho en publicarse

Desde la versión 9, se publica una nueva versión cada 6 meses

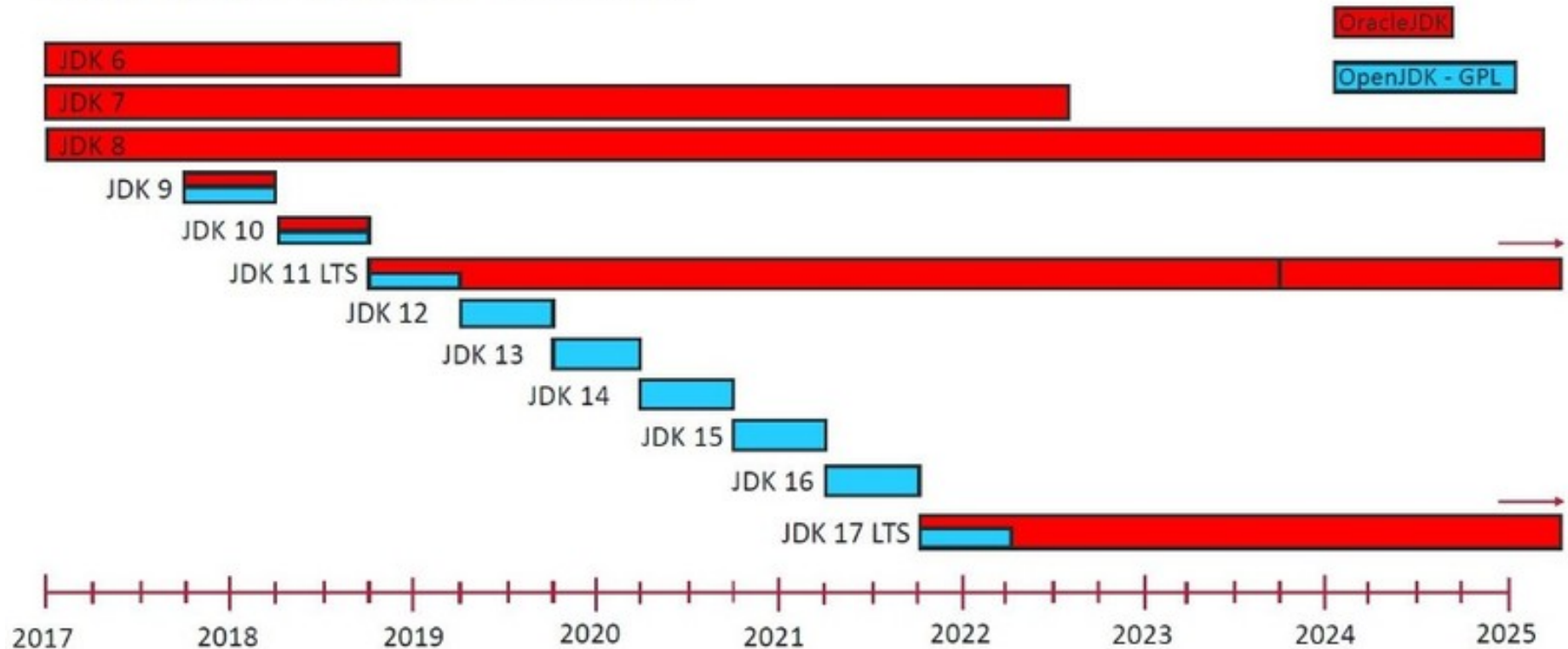
Evolución de Java

Java Release and Support Cadence: Java 4 through 13



Versiones LTS

New Java Release Timeline*

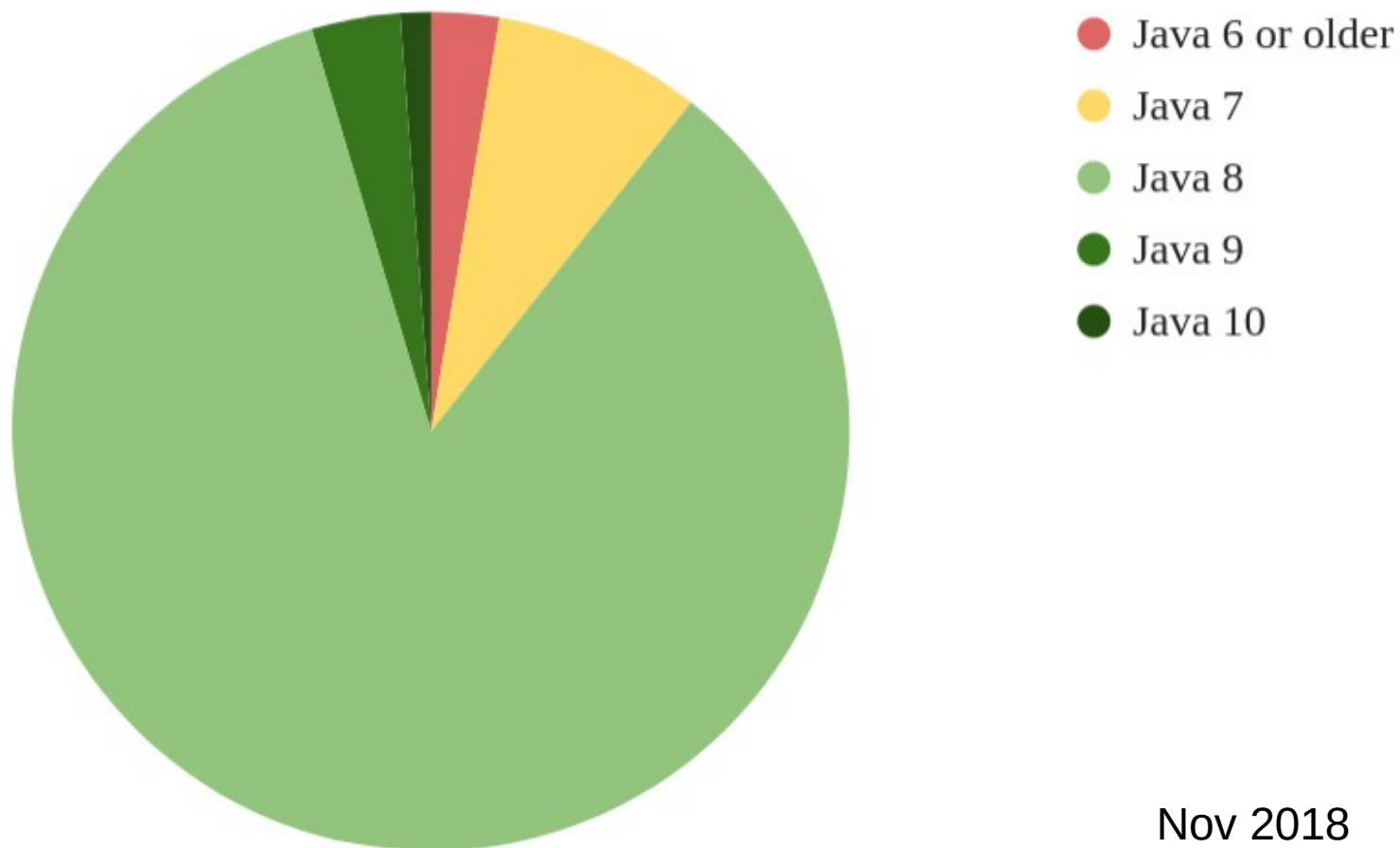


* Dates and version numbers subject to change

LTS = Long term support release

Qué versión se usa hoy?

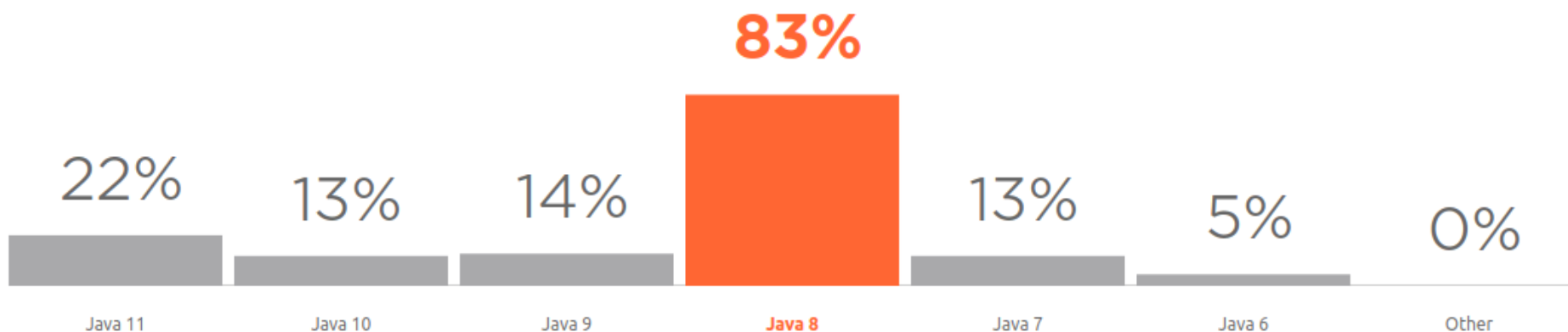
Java Adoption in 2018



Nov 2018

Qué versión se usa hoy?

Which versions of Java do you regularly use?



Agosto 2019

<https://www.jetbrains.com/lp/devecosystem-2019/java/>

- El código fuente de Java es libre (licencia GPL)
- El **binario oficial distribuido por Oracle** es comercial pasados unos meses desde su publicación
- Han aparecido “**distribuciones**” de **Java** alternativas a la oficial

Amazon corretto



<https://aws.amazon.com/es/corretto/>

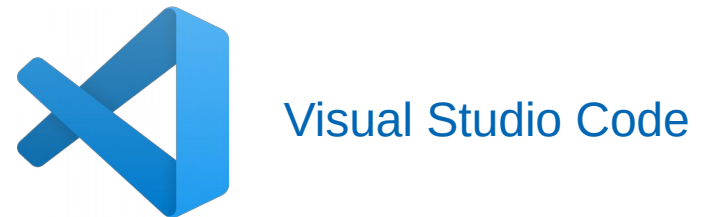
Azul Zulu Community



<https://www.azul.com/downloads/zulu-community/>

Linux distros





- **Maven**

- Tecnología más usada para definir proyectos Java
- Sistema de **gestión de dependencias** (*librerías*) y sus versiones
- Sistema de **construcción de proyectos** (de código a entregable .zip)
- Estructura única de proyecto compatible con todos los **entornos de desarrollo** y sistemas de **integración continua**

- **Cómo crear un proyecto Maven en Eclipse**
 - File > New > Maven Project
 - Dejar la plantilla que aparece por defecto seleccionada (**maven-archetype-quickstart**)
 - Indicar el nombre del proyecto en dos partes:
 - **GroupId: es.codeurjc.app**
 - **ArtifactId: helloworld**

Maven

New Maven Project

New Maven project
Specify Archetype parameters

Group Id: es.codeurjc.app

Artifact Id: helloworld

Version: 0.0.1-SNAPSHOT

Package: es.codeurjc.app.helloworld

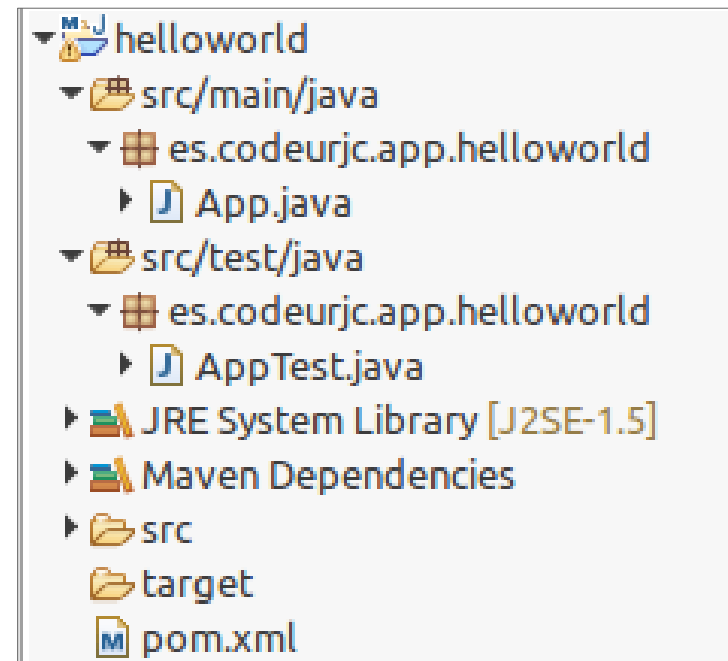
Properties available from archetype:

Name	Value
------	-------

Advanced

< Back Next > Cancel Finish

- Los proyectos Maven tienen la siguiente **estructura**
 - **src/main/java**: Código de la aplicación
 - **src/test/java**: Código de los tests
 - **pom.xml**: Fichero de descripción del proyecto (nombre, dependencias, configuraciones, etc...)



- **pom.xml**: Configuración del proyecto

The screenshot displays the Maven IDE interface for a project named 'helloworld'. The 'Overview' tab is active, showing the following configuration:

- Artifact**
 - Group Id: `es.codeurjc.app`
 - Artifact Id: `*helloworld`
 - Version: `0.0.1-SNAPSHOT`
 - Packaging: `jar`
- Parent** (with icons for search and folder)
- Properties**
 - `<code>project.build.sourceEncoding : UTF-8</code>`
 - Buttons: `Create...`, `Remove`
- Modules** (with button: `New module element`)
- Project**
 - Name: `helloworld`
 - URL: `http://maven.apache.org`
 - Description: (empty text area)
 - Inception: (empty text area)
- Other Sections**
 - `Organization`
 - `SCM`
 - `Issue Management`
 - `Continuous Integration`

The bottom of the interface shows a tabbed view with the following tabs: `Overview`, `Dependencies`, `Dependency Hierarchy`, `Effective POM`, and `pom.xml`.

- **pom.xml**: Configuración del proyecto



```
<?xml xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>es.codeurjc.app</groupId>
  <artifactId>helloworld</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>helloworld</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Poner la vista de código fuente

Overview Dependencies Dependency Hierarchy Effective POM pom.xml

- **pom.xml:** Configuración del proyecto
 - **groupId:** Organización, familia
 - **artifactId:** Nombre del proyecto
 - **version:** Versión del proyecto (especialmente útil para librerías)
 - **packaging:** Tipo de aplicación (jar es una aplicación normal)
 - **name:** Nombre “bonito” del proyecto (para documentación)
 - **url:** Web del proyecto (para documentación)

```
<groupId>es.codeurjc.app</groupId>  
<artifactId>helloworld</artifactId>  
<version>0.0.1-SNAPSHOT</version>  
<packaging>jar</packaging>
```


- **pom.xml:** Configuración del proyecto
 - **properties:**
 - ▮ Configuraciones generales del proyecto
 - ▮ Codificación de los ficheros fuente

```
<properties>  
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
</properties>
```

- **pom.xml: Configuración del proyecto**
 - **dependencies:**
 - ▮ Dependencias (**librerías**)
 - ▮ Cada librería está identificada por su **groupId**, **artifactId** y **versión** (coordenadas)
 - ▮ Se pueden poner tantas dependencias como se quiera

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

- Actualizamos el pom.xml para usar versiones más recientes de **Java (1.8)**
- **properties**

```
<properties>  
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
  <maven.compiler.source>1.8</maven.compiler.source>  
  <maven.compiler.target>1.8</maven.compiler.target>  
</properties>
```

- **pom.xml:** Configuración del proyecto
 - **Cuidado!** Algunos cambios en el fichero pom.xml no se reflejan en eclipse de forma automática
 - Cuando se hace un cambio y eclipse no se actualiza con esos cambios, se tiene que indicar explícitamente
 - ▮ Botón derecho proyecto > Maven > Update Project...

- **Importar proyectos Maven en eclipse**
 - Si está dentro de un .zip, descomprimir
 - File > Import > Existing Maven project
 - Seleccionar la carpeta raíz de todos los proyectos

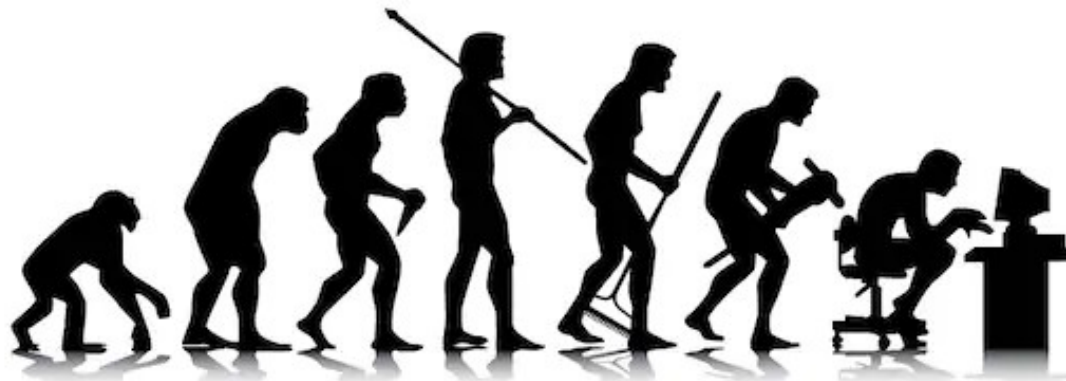
Cambios desde Java 6

- Desde Java 6 hasta Java 13 en Java ha habido **muchos cambios, pero mantiene su espíritu**
- Focalizaremos en los cambios de **Java 6 a 8**
- Presentaremos de forma rápida los cambios de **Java 8 a 13**



¿Cómo evoluciona Java?

- Lenguaje de programación ←
- Librería estándar (API) ←
- Java Virtual Machine
- Herramientas, plugins...



- Se han añadido mejoras en la **productividad del desarrollador**
 - Cómo hacer lo mismo pero sin tanta repetición, con código más **conciso**
- Se han hecho cambios para una programación **más funcional** (expresiones lambda)
 - El principal sigue siendo el paradigma **orientado a objetos**

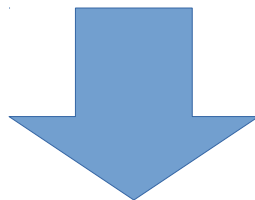


Lenguaje

Diamond Operator (Java 7)

- Evitar repetición en declaración de variables y atributos genéricos

```
Map<String, List<Trade>> trades = new TreeMap<String, List<Trade>>();
```



```
Map<String, List<Trade>> trades = new TreeMap<>();
```

Strings en switch (Java 7)

```
1 String dayTime = "evening";
2 switch (dayTime) {
3     case "morning":
4         System.out.println("Good morning!");
5         break;
6     case "noon":
7         System.out.println("Good afternoon!");
8         break;
9     case "evening":
10        System.out.println("Good evening!");
11        break;
12 }
```

Gestión automática de recursos (Java 7)

- **Try-with-resources:** Los recursos se cierran después de usarse

```
1 try (FileInputStream in = new FileInputStream("properties.txt")){  
2     System.out.println(in.read());  
3 }
```

- Cualquier clase que implemente el interfaz **AutoCloseable** se puede usar con el try

```
1 public class Deur implements AutoCloseable {  
2  
3     @Override  
4     public void close() throws IOException {  
5         System.out.println("Deur toe");  
6     }  
7  
8     public void openDeur() {  
9         System.out.println("Deur is open");  
10    }  
11 }  
12 }
```

```
1 try(Deur deur = new Deur()){  
2     deur.openDeur();  
3 }
```

Gestión automática de recursos (Java 7)

Gestionar dos ficheros para copiar el contenido de uno a otro

```
3
4 // Java 6
5 InputStream in = null;
6 OutputStream out = null;
7 try {
8     in = new FileInputStream(file);
9     // Read from file
10    try {
11        out = new FileOutputStream(file2);
12        // Write to file
13    } catch (IOException e) {
14    } finally {
15        try {
16            out.close();
17        } catch (IOException e) {
18        }
19    }
20 } catch (IOException e) {
21 } finally {
22    try {
23        in.close();
24    } catch (IOException e) {
25    }
26 }
27
```

```
27
28 // Java 7
29 try (InputStream in2 = new FileInputStream(file);
30      OutputStream out2 = new FileOutputStream(file2)) {
31    // Read and write
32 } catch (IOException ex) {
33    // Handle Exception
34 }
```

Gestión de excepciones mejorada (Java 7)

```
1 // Java 6
2 try {
3     interestingMethod();
4 } catch (ClassNotFoundException ex) {
5     // Handle Exception
6 } catch (NoSuchMethodException ex) {
7     // Handle Exception
8 } catch (NoSuchFieldException ex) {
9     // Handle Exception
10 }
```

```
11
12 // Java 7
13 try {
14     interestingMethod();
15 } catch (ClassNotFoundException | NoSuchMethodException | NoSuchFieldException ex) {
16     // Handle Exception
17 }
```

Rethrow más preciso (Java 7)

```
public void process() throws Exception {  
    try {  
        int error = work();  
  
        if(error == 1) {  
            throw new FirstException();  
        } else if (error == 2)  
            throw new SecondException();  
        }  
  
    } catch (Exception e) {  
        freeResources();  
        throw e;  
    }  
}
```

Como en el catch
se captura
Exception, en
Java 6 se pone
en el método
Exception

Rethrow más preciso (Java 7)

```
public void process() throws  
    FirstException, SecondException {  
  
    try {  
  
        int error = work();  
  
        if(error == 1) {  
            throw new FirstException();  
        } else if (error == 2)  
            throw new SecondException();  
        }  
  
    } catch (Exception e) {  
        freeResources();  
        throw e;  
    }  
}
```

Desde Java 7, se permite indicar las excepciones que realmente se lanzan, no el tipo de la variable donde se capturan

Mejoras en números literales (Java 7)

- Binarios literales con 0b o 0B
- Separadores numéricos con guión bajo

```
long creditCardNumber = 1234_5678_9012_3456L;  
long socialSecurityNumber = 999_99_9999L;  
float pi = 3.14_15F;  
long hexBytes = 0xFF_EC_DE_5E;  
long hexWords = 0xCAFE_BABE;  
long maxLong = 0x7fff_ffff_ffff_ffffL;  
byte nybbles = 0b0010_0101;  
long bytes = 0b11010010_01101001_10010100_10010010;
```



API

Nueva API de Ficheros (Java 7)

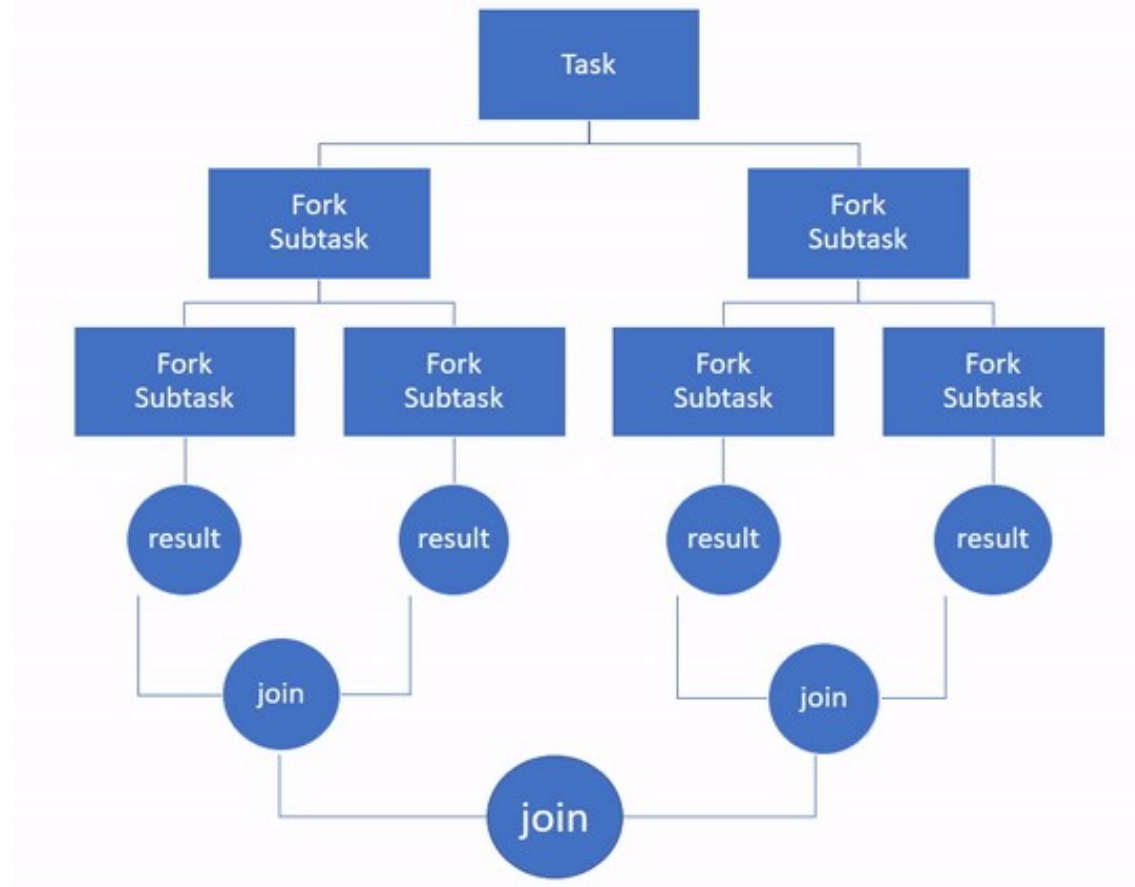
- La clase **java.io.File** representa tanto una ruta como un fichero en disco
- Desde Java 7, para manipular rutas se usa la clase **java.nio.Path** y para manipular el contenido del disco se usan los métodos estáticos de la clase **java.nio.Files**

Fork Join (Java 7)

- Framework para paralelizar un algoritmo
- Consiste en dividir una tarea en tareas más pequeñas (de forma recursiva) hasta tareas atómicas
- Las tareas atómicas se ejecutan en paralelo

<https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-io>

Fork Join (Java 7)



<https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html>

Nueva API de Ficheros (Java 7)

- La clase **java.io.File** representa tanto una ruta como un fichero en disco
- Desde Java 7, para manipular rutas se usa la clase **java.nio.Path** y para manipular el contenido del disco se usan los métodos estáticos de la clase **java.nio.Files**

- **Path**

- Existen métodos para manipular rutas en disco

```
Path p = Paths.get("/articles/baeldung");
```

```
String fileName = ...
```

```
Path p = Paths.get("/articles", fileName);
```

- **Path**

- Existen métodos para manipular rutas en disco

```
Path p = Paths.get("/articles/baeldung/logs");  
Path fileName = p.getFileName();  
System.out.print(fileName.toString());  
//Prints logs
```


Nueva API de Ficheros (Java 7)

- **Path**

- Existen métodos para manipular rutas en disco

```
Path p1 = Paths.get("/articles/baeldung/logs");  
Path parent = p1.getParent();  
System.out.print(parent.toString());  
//Prints \articles\baeldung
```

Nueva API de Ficheros (Java 7)

- **Path**

- Existen métodos para manipular rutas en disco

```
Path p = Paths.get("/home/user/../articles");  
Path cleanPath = p.normalize();  
System.out.print(cleanPath.toString());  
// Prints \\home\\articles
```

Nueva API de Ficheros (Java 7)

- **Files**

- Manipula ficheros en disco (representados por Path)
- Errores lanzan excepciones

```
Path p = Paths.get("file.txt");  
boolean exists = Files.exists(p);  
boolean directory = Files.isDirectory(p);
```

- Readable, writable, executable, regularFile...

Nueva API de Ficheros (Java 7)

- **Files**

```
Path p = Paths.get("file.txt");  
Files.createFile(p);  
Files.delete(p);  
Path p = Paths.get("/home/pepe/data");  
Files.createDirectories(p);
```

<https://www.baeldung.com/java-nio-2-file-api>

Nueva API de Ficheros (Java 7)

- **Files**

- Ficheros temporales

```
String prefix = "log_";  
String suffix = ".txt";  
Path p = Paths.get("/home/user");  
Files.createTempFile(p, prefix, suffix);
```

Nueva API de Ficheros (Java 7)

- **Files**

- Lista contenidos de una carpeta

```
Path dir = ...  
  
try (DirectoryStream<Path> dirStream =  
    Files.newDirectoryStream(dir)) {  
    for (Path path: dirStream) {  
        System.out.println(path);  
    }  
}
```

<https://www.baeldung.com/java-list-directory-files>

- **Files**

- Lista contenidos de una carpeta (recursivamente)

```
Path dir = ...

Files.walkFileTree(dir, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path path,
        BasicFileAttributes attrs) throws IOException {
        System.out.println(path);
        return FileVisitResult.CONTINUE;
    }
});
```

<https://www.baeldung.com/java-list-directory-files>

- **Files**

- Copiar ficheros

```
Path file1 = ...  
Path file2 = ...  
Files.copy(file1, file2);  
Files.copy(file1, file2,  
    StandardCopyOption.REPLACE_EXISTING);
```

<https://www.baeldung.com/java-nio-2-file-api>

- **Files**

- Mover ficheros

```
Path file1 = ...  
Path file2 = ...  
Files.move(file1, file2);  
Files.move(file1, file2,  
    StandardCopyOption.REPLACE_EXISTING);
```

Nueva API de Ficheros (Java 7)

- **Files**

- Leer ficheros

```
Path file = ...  
  
byte[] fileBytes = Files.readAllBytes(file);  
  
String data = new String(fileBytes);
```

Nueva API de Ficheros (Java 7)

- **Files**
 - Leer ficheros

```
Path file = ...  
  
List<String> lines = Files.readAllLines(file);
```

- **Files**

- Leer ficheros

```
Path logFile = ...

try (BufferedReader reader =
    Files.newBufferedReader(logFile, StandardCharsets.UTF_8)){

    String line;

    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }

} catch (Exception e) {
    ...
}
```

Nueva API de Ficheros (Java 7)

- **Files**

- Escribir ficheros

```
Path file = ...  
  
String data = ...  
  
Files.write(path, data.getBytes());
```

Nueva API de Ficheros (Java 7)

- **Files**

```
try {  
    Path file = ...  
    Files.createFile(file);  
    try (PrintWriter pw =  
        new PrintWriter(Files.newBufferedWriter(p))) {  
        for (RecentFile recentFile : list) {  
            pw.println(recentFile);  
        }  
    }  
} catch (IOException e) {  
    ...  
}
```

Ejercicio 1

- Implementa un programa Java que escriba en un fichero de texto (**files.txt**) los nombres de todos los ficheros y directorios dentro de un directorio especificado
- Se podrá parametrizar para que las rutas sean **absolutas o relativas** a la carpeta indicada
- Se podrá parametrizar para que solo escriba los ficheros con un prefijo **determinado**



Lenguaje

- Cuando se desarrolla software es habitual la **reutilización de código** en forma de métodos (o funciones)
- Esos métodos se **adaptan** para el caso concreto **mediante parámetros**
- Los parámetros suelen ser **datos**

Parámetro



```
System.out.println("Texto a mostrar por pantalla");
```

- En ocasiones resulta útil pasar **código** como **parámetro**
- Por ejemplo:
 - Obtener la lista de ficheros dentro de un directorio que cumplen con un determinado **filtro**.
 - Ese **filtro** es un código que se ejecutará para cada fichero y devolverá **true** si el fichero pasa el filtro o **false** si el fichero no pasa el filtro.

- En la librería de Java la clase **File** permite listar el contenido de una carpeta pasando un **filtro (código)** como parámetro.

```
public String[] list(FilenameFilter filter)
```

- ¿Cómo se llama a este método?
- ¿Cómo se le pasa el código del filtro?

- Se puede pasar un **objeto** de una clase que implemente el interfaz **FilenameFilter**:

```
class TextFilesFilter implements FilenameFilter {  
  
    @Override  
    public boolean accept(File dir, String name) {  
        return name.endsWith(".txt");  
    }  
}
```

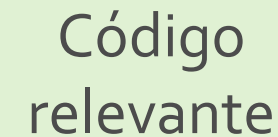
```
File currentDir = new File("");  
  
String[] files = currentDir.list(new TextFilesFilter());
```

Expresiones Lambda

- Este código es **demasiado largo** para la funcionalidad que se desea

```
class TextFilesFilter implements FilenameFilter {  
  
    @Override  
    public boolean accept(File dir, String name) {  
        return name.endsWith(".txt");  
    }  
}
```

Código
relevante




```
File currentDir = new File("");  
  
String[] files = currentDir.list(new TextFilesFilter());
```

- Las **expresiones lambda** son una forma **compacta** de pasar a un método **código** como **parámetro**

Expresión lambda

```
String[] files =  
    currentDir.list((dir,name) -> name.endsWith(".txt"));
```



- Se escribe únicamente lo **relevante**

- ¿Cuándo se puede usar una expresión lambda?
 - En todos aquellos lugares en los que se espere un objeto que implementa un **interfaz con un único método (FunctionalInterface)**

Interfaz con un método



```
FilenameFilter filter = (dir,name)->name.endsWith(".txt");  
String[] files = currentDir.list(filter);
```

- **¿Cómo se escribe una expresión lambda?**
 - Si el método no tiene parámetros se ponen los paréntesis

```
Runnable runnable = () -> System.out.println();
```

- Si sólo tiene un parámetro, se pueden omitir

```
FileFilter filter = f -> f.isDirectory();
```


- ¿Cómo se escribe una expresión lambda?

- Si el método tiene que devolver un valor y hay una única sentencia, se omite el **return**

```
FileFilter filter = f -> f.isDirectory();
```

- Si son varias sentencias, se ponen entre **{ }** y se pone el **return**

```
FileFilter filter = f -> {  
    int numBytes = f.length();  
    return numBytes > 100;  
}
```

- ¿Cómo se escribe una expresión lambda?
 - Si lo único que vas a hacer es invocar un método en el objeto que te pasan como parámetro, puedes usar una **referencia a método (Method reference)**

```
FileFilter filter = f -> f.isDirectory();
```



```
FileFilter filter = File::isDirectory;
```

- ¿Cómo se escribe una expresión lambda?
 - Si quieres, puedes poner el **tipo de los parámetros**, pero no es necesario

```
FileFilter filter = f -> f.isDirectory();
```



```
FileFilter filter = (File f) -> f.isDirectory();
```

- ¿Qué información puedo usar en una lambda?
 - Puedes usar los **parámetros** de la lambda
 - Puedes usar **atributos** de la clase en la que se escribe
 - Puedes usar **this**, y se refiere al objeto en el que se escribe
 - Puedes usar **variables locales**, pero no pueden cambiar de valor

```
String ext = "txt";  
file.list((dir,name)->name.endsWith(".") + ext));
```

Variable local



- **Clases anónimas**

- En versiones anteriores de Java, la forma habitual de pasar código como parámetro era usar las **clases anónimas**
- Las clases anónimas son una **forma compacta** de implementar una **clase completa** dentro de un método
- En la misma sentencia se **declara la clase** y se **crea un objeto** de dicha clase
- Una clase anónima **no tiene nombre**

- Clases anónimas

```
public static void main(String[] args) {  
  
    Runnable r = new Runnable() {  
        @Override  
        public void run() {  
            System.out.println("Hola mundo");  
        }  
    };  
}
```

- Clases anónimas

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

```
    Runnable r = new Runnable() {  
        @Override  
        public void run() {  
            System.out.println("HoLa Mundo");  
        }  
    };  
}
```

Se indica el interfaz que implementa la clase anónima (o la clase padre)

Se crea un objeto de la clase y se asigna a una variable

Cuerpo de la clase entre llaves

- **Clases anónimas**
 - Las clases anónimas son muy **largas de escribir**
 - Ahora es **preferible usar una expresión lambda** siempre que sea posible en vez de una clase anónima

- Clases anónimas

```
public static void main(String[] args) {  
  
    Runnable r = new Runnable() {  
        @Override  
        public void run() {  
            System.out.println("Hola mundo");  
        }  
    };  
}
```



```
public static void main(String[] args) {  
    Runnable r = () -> System.out.println("Hola mundo");  
}
```

- **Clases anónimas vs Expresiones lambda**
 - Las expresiones lambda son más **eficientes** en tiempo de ejecución que las clases anónimas y son **más compactas** de escribir
 - Una clase anónima es una **clase completa** (puede tener clase padre, atributos, implementar varios métodos...).
 - Una lambda sólo puede usarse donde se espere un **interfaz con un método**

- **Clases anónimas vs Expresiones lambda**
 - Si se usa **this** en una clase anónima, se referencia al **objeto de la clase anónima**
 - Si se usa **this** en una expresión lambda, se referencia al **objeto donde se está declarando la lambda**

- Un lenguaje de programación se puede considerar **funcional** si permite manejar **funciones**:
 - Asignar funciones a variables
 - Pasar funciones como parámetro
 - Definir funciones anónimas en los mismo lugares que se puede poner un valor o una expresión
- Hay muchos lenguajes imperativos que permiten manejar funciones: **JavaScript, Python, Ruby, Groovy, Scala, C#, C++...**

- Ejemplo de función en JavaScript

```
function processParams(param1, param2, callback) {  
    alert('Hello: '+param1+', '+param2);  
    callback();  
}  
  
processParams('ham', 'cheese', function() {  
    alert('Finished eating my sandwich.');});
```

- Cuando se define una función usando una expresión (directamente en el código) se denomina *lambda* o *clousure* (cerradura)

Ejercicio 2

- Refactoriza el programa del ejercicio 1 para que exista una clase **DirectoryToFile** (que podrá ser usada en otros contextos)
- La clase tendrá un método **processDir** que recibirá como parámetro un Path y un booleano sobre las rutas relativas y generará el fichero
- Tendrá otro método sobrecargado que además recibirá una **expresión lambda** que permitirá filtrar los ficheros que se muestran en base al path del fichero

Ejercicio 2

```
public class DirectoryToTextFile {  
    public void processDir(Path path, boolean relative){...}  
    public void processDir(Path path, boolean relative,  
        PathFilter filter){...}  
}
```

```
public interface PathFilter {  
    ...  
}
```

Extension methods

- Implementación de métodos por defecto en interfaces

```
1 interface Printer {  
2     default void print(String s) {  
3         System.out.println(s);  
4     }  
5 }
```

- Permite **reutilizar una implementación de método** definida en el interfaz sin los problemas de la **herencia múltiple** (debidos a la duplicación de atributos)

- ¿Qué ocurre una clase implementa **dos interfaces** que proporcionan implementaciones para el mismo método?
- La clase tiene que **implementar el método para desambiguar**

```
static class MultifunctionalPrinter implements Printer, Copier {  
    @Override  
    public void print(String s) {  
        Copier.super.print(s);  
    }  
}
```



API

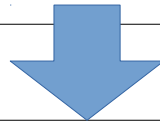
- Los default methods en interfaces permiten añadir **métodos a las estructuras de datos** (interfaces List, Set y Map)
- Implementaciones de la clase Collections se han implementado en las interfaces

```
List<Integer> nums = ...  
nums.sort(Comparator.naturalOrder());
```

- Métodos para procesamiento funcional

```
List<Integer> nums = ...  
nums.forEach(System.out::println);
```

```
Set<String> value = map.get(key);  
if (value == null) {  
    value = new HashSet<>();  
    map.put(key, value);  
}  
value.add(v);
```



```
map.computeIfAbsent(key, k -> new HashSet<V>()).add(v);
```

Functional Interfaces

- Anotación `@FunctionalInterface`
- Es una anotación que sirve para **comunicar** que el interfaz está diseñado para que use con una expresión lambda (sólo un método)
- No es obligatorio

```
1  @FunctionalInterface
2  interface Formula {
3      double calculate(int a);
4  }
```

- La API incluye varios Functional Interfaces genéricos que se usan en diferentes métodos de la API
 - `Predicate<T>: boolean test(T t);`
 - `Function<T>: R apply(T t);`
 - `Supplier<T>: T get();`
 - `Consumer<T>: void accept(T t);`
 - `Comparator<T>: int compare(T o1, T o2);`
- Paquete `java.util.function`

- Los **streams** (flujos) permiten procesar colecciones de elementos con un estilo **funcional (declarativo)**
- **Ventajas** frente al estilo actual:
 - Más **compacto** y de **alto nivel**
 - Fácilmente **paralelizable**
 - No se necesita tener todos los elementos en **memoria** para empezar a procesar

- Tenemos la clase **Person** y la lista **people**

```
public class Person {  
  
    private String name;  
    private int age;  
  
    // constructors  
    // getters / setters  
}
```

```
List<Person> people = new ArrayList<>();
```


- Se quiere calcular la edad media

```
int sum = 0;
int average = 0;
for (Person person : people) {
    sum += person.getAge();
}

if (!list.isEmpty()) {
    average = sum / list.size();
}
```

- Se quiere calcular la edad media de los menores de 20

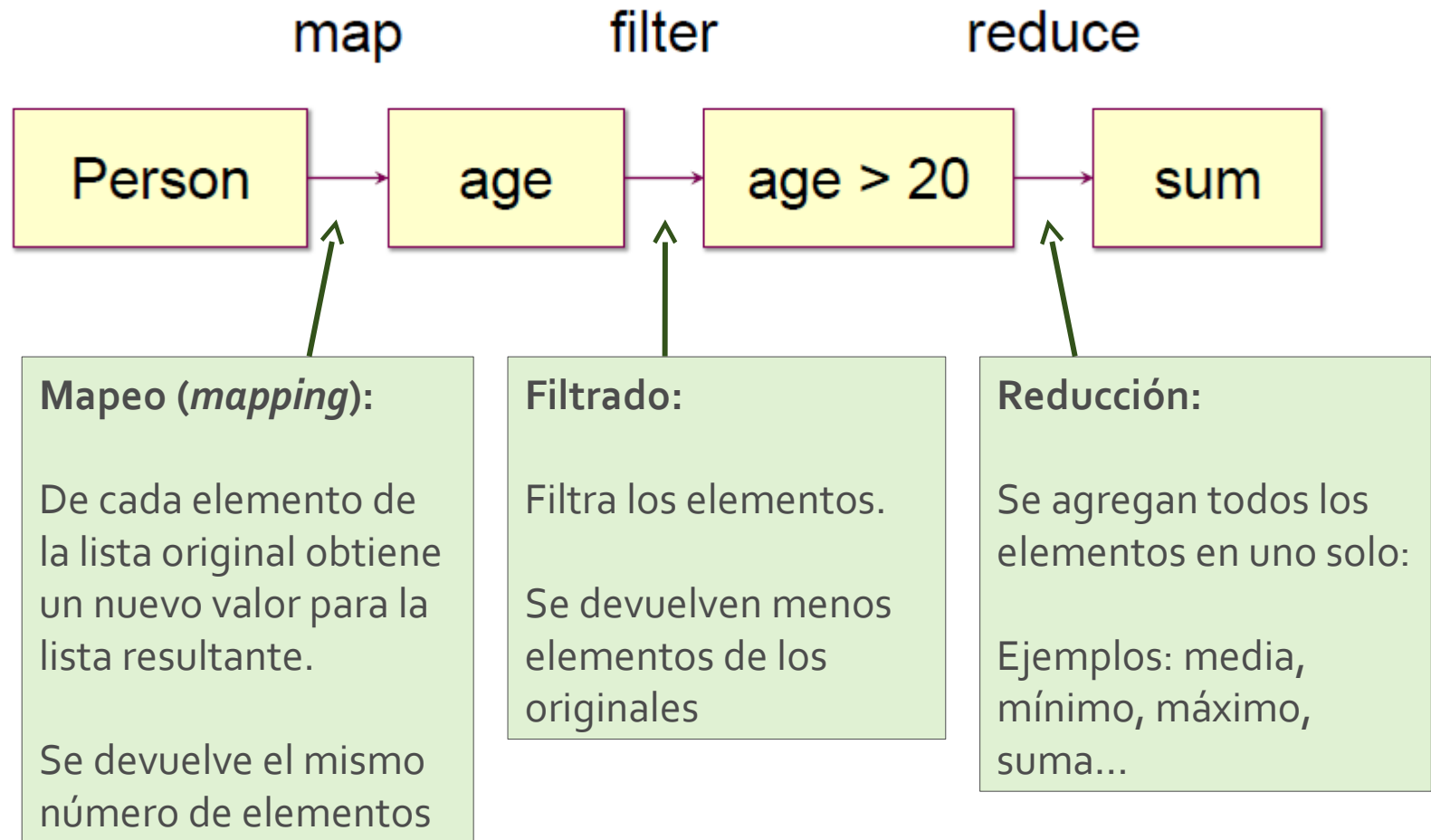
```
int sum = 0;
int n = 0;
int average = 0;
for (Person person : people) {
    if (person.getAge() > 20) {
        n++;
        sum += person.getAge() ;
    }
}
if (n > 0) {
    average = sum / n ;
}
```

- La **programación imperativa** es muy *verbosa* porque se indica **qué** se quiere y **cómo** se consigue
- La **programación declarativa** es mucho más compacta porque basta indicar **qué** se quiere

```
select avg(age)
from Person
where age > 20
```

Sentencia SQL equivalente

Streams



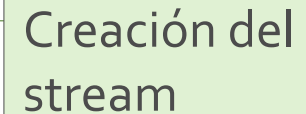
- Los *streams* (**flujos**) permiten aplicar operaciones de mapeo, filtrado y reducción a colecciones de datos
- Un stream es **una secuencia de elementos** que sólo pueden **procesarse una vez**
- Puede **generarse** de forma dinámica (no tiene que estar toda la secuencia en memoria)
- Esto permite **implementaciones muy eficientes** de las operaciones **sin** crear estructuras de **datos intermedias**

- La forma más habitual de crear un **stream** es partiendo de una **colección**

```
List<Person> persons = ...
```

```
int sumOver20 = persons.stream()  
    .map(Person::getAge)  
    .filter(age -> age > 20)  
    .reduce(0, Integer::sum);
```

Creación del
stream



- También se pueden crear de forma literal:

```
Stream<String> s = Stream.of("one", "two", "three");  
Stream<String> s2 = Stream.empty();
```

- Partiendo de un array:

```
String[] numbers = {"one", "two", "three"};  
Stream<String> s = Arrays.stream(numbers);
```

- O desde algunos métodos de la API:

```
IntStream chars = "Hola".chars();  
  
Stream<String> lines = lineNumberReader.lines();  
  
IntStream nums = random.ints();  
  
Stream s3 = Stream.concat(stream1, stream2);  
  
Stream<Integer> pares = Stream.iterate(0, n->n+2);  
  
Stream<Double> rand = Stream.generate(Math::random);
```


Stream	Colección
Secuencia de elementos	Elementos que se pueden procesar en secuencia o con acceso directo (get)
Se pueden calcular según se van procesando	Tienen que estar en memoria antes de procesarse
Información volátil que sólo se puede procesar una vez	Estructuras de datos en memoria
Tamaño infinito	Tamaño finito
Tiene versiones eficientes para tipos primitivos (IntStream, DoubleStream, LongStream)	No tiene versiones para tipos primitivos

- **Operaciones** que se pueden realizar sobre un stream:
 - 1) Operaciones **intermedias**
 - ▢ Se procesan de forma perezosa (sólo si son necesarias)
 - ▢ Puede haber varias (incluso del mismo tipo)
 - ▢ Ejemplos: map, filter, skip,...
 - 2) Operaciones **terminales**
 - ▢ Inician la computación y devuelven un objeto, un valor, una lista o nada...
 - ▢ Sólo puede haber una al final
 - ▢ Una vez aplicada, el stream no se puede reutilizar
 - ▢ Ejemplos: sum, find, min, toArray...

- **Operaciones** que se pueden realizar sobre un stream:

```
List<Person> persons = ...
```

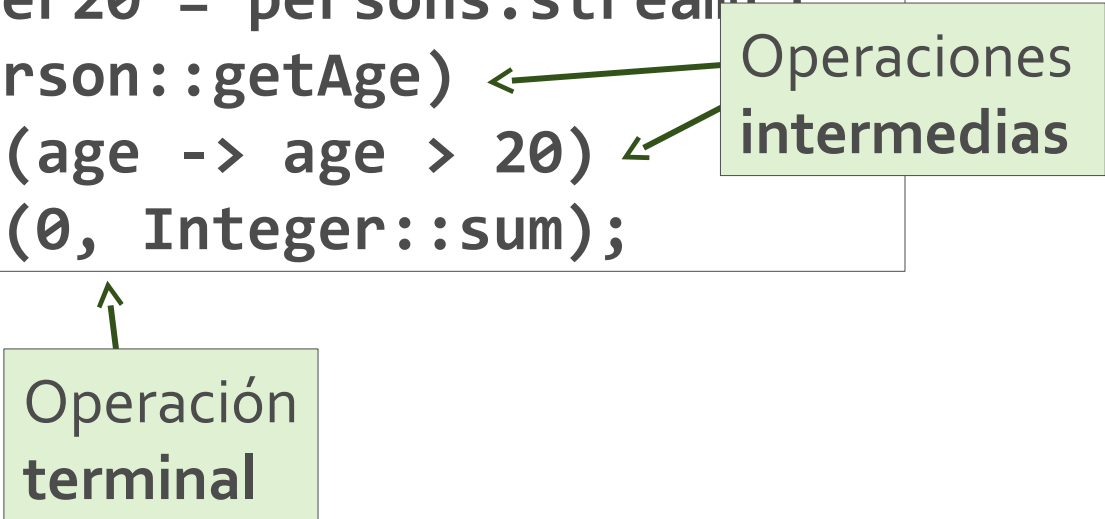
```
int sumOver20 = persons.stream()
```

```
    .map(Person::getAge)
```

```
    .filter(age -> age > 20)
```

```
    .reduce(0, Integer::sum);
```

Operaciones
intermedias



Operación
terminal

- **Operaciones intermedias:**
 - **filter:** quita algunos elementos
 - **map:** por cada elemento obtiene un nuevo valor:
 - **sorted:** ordena
 - **peek:** aplica una operación a cada elemento
 - **distinct:** filtra dejando los distintos
 - **limit:** limita el número de elementos
 - **skip:** ignora los primeros elementos
 - **range:** devuelve un rango de los elementos (from, to)

- **Operaciones terminales en Stream<T>:**
 - **count:** Cuenta los elementos.
 - **min, max:** Obtiene el mínimo el y máximo
 - **anyMatch, allMatch, noneMatch():** Indica si se cumple (o no) el criterio.
 - **findFirst, findAny:** Devuelve el elemento que cumpla el criterio
 - **mapToInt:** Convierte a IntStream
 - **toArray:** Devuelve el contenido como array
 - **forEach, forEachOrdered:** Ejecuta por cada elemento
 - **reduce:** Mecanismo genérico de reducción de todos los elementos
 - **collect:** Mecanismo genérico para “recolectar” los elementos

- **Operaciones terminales en streams numéricos (IntStream, LongStream, DoubleStream):**
 - **average():** Calcula la media
 - **sum():** Suma los elementos
 - **summaryStatistics():** Calcula estadísticas de los datos (media, cuenta, min, max, sum)

- **Operación terminal Collect (Recolectar)**
 - La operación **collect** es un mecanismo genérico para implementar operaciones terminales
 - Se puede implementar **cualquier** política de recolección de los elementos:
 - Devolver una estructura de datos (List, Set, Map...)
 - Fusionar todos los elementos en un String
 - Obtener un valor (suma, media, min, max...)
 - La clase **Collectors** tiene métodos estáticos para crear una infinidad de **recolectores**

```
ArrayList<String> l = stream.collect(Collectors.toList());
```

- Edad media

```
double avgAge = persons.stream()  
    .collect(Collectors.averagingDouble(Person::getAge));
```

- Número de personas mayores de 20

```
int num = persons.stream().filter(p -> p.getAge() > 20)  
    .collect(Collectors.counting());
```

- Persona más mayor (si existen)

```
Optional<Person> older = persons.stream()  
    .collect(Collectors.maxBy(Comparator.comparing(Person::getAge)));
```


- Devolver los nombres en una lista

```
List<String> list = people.stream().map(Person::getName)
    .collect(Collectors.toList());
```

- Devolver los nombres sin repeticiones ordenados (TreeSet)

```
Set<String> set = people.stream().map(Person::getName)
    .collect(Collectors.toCollection(TreeSet::new));
```

- Devolver los nombres separados por comas

```
String joined = things.stream().map(Object::toString)
    .collect(Collectors.joining(", "));
```

- Suma de salarios de los empleados

```
int total = employees.stream()  
    .collect(Collectors.summingInt(Employee::getSalary));
```

- Agrupar por departamento

```
Map<Department, List<Employee>> byDept = employees.stream()  
    .collect(Collectors.groupingBy(Employee::getDepartment));
```

- Suma de salarios por departamento

```
Map<Department, Integer> totalByDept = employees.stream()  
    .collect(Collectors.groupingBy(  
        Employee::getDepartment,  
        Collectors.summingInt(Employee::getSalary)));
```

- Suma de salarios por departamento

```
Map<Department, Integer> totalByDept = employees.stream()
    .collect(
        Collectors.groupingBy(
            Employee::getDepartment,
            Collectors.summingInt(Employee::getSalary)
        )
    );
```

- Dividir los estudiantes entre aprobados y suspensos

```
Map<Boolean, List<Student>> passingFailing = students.stream()
    .collect(
        Collectors.partitioningBy(s->s.getGrade() >= PASS_THR)
    );
```

- La principal ventaja de los streams es que especificamos **las operaciones** que queremos que se realicen, pero no especificamos **cómo deben realizarse**.
- **Por defecto** las operaciones se ejecutan de **forma secuencial** en el hilo de ejecución, pero podemos pedir que se ejecuten en **paralelo**

- Partiendo de un stream se puede obtener su **stream paralelo**
- Todas las operaciones que se ejecutarán de forma automática en paralelo, **dividiendo** las tareas en **diferentes hilos**

```
Stream<String> s = nombres.parallelStream();
```

```
Stream<String> s = nombres.stream().parallel();
```

Ejercicio 3

- Partiendo de una lista de enteros, mostrar por pantalla:
 - La suma y la cuenta de los números pares mayores de 10
 - La suma y la cuenta de los números impares mayores de 10
- Implementar con y sin streams

- Limitaciones de gestión de fechas en Java <8
 - Seguridad ante hilos (ThreadSafe)
 - Mutabilidad
 - Diseño de la API farragoso y propenso a errores
 - Las fechas con TimeZone no se gestionan adecuadamente

- Instant
 - Tiempo con precisión de nanosegundos

```
Instant instant = Instant.now();  
  
// Output format is ISO-8601  
System.out.println(instant);  
  
instant = Instant.ofEpochMilli(new Date().getTime());  
  
instant = Instant.parse("1995-10-23T10:12:35Z");
```


- Instant
 - Manipulación

```
instant.plus(Duration.ofHours(5).plusMinutes(4));
```

```
instant.minus(5, ChronoUnit.DAYS); // Option 1
```

```
instant.minus(Duration.ofDays(5)); // Option 2
```

```
boolean after = instant1.isAfter(instant);
```

```
boolean before = instant1.isBefore(instant);
```

- Fechas y horas locales (sin TimeZone)
 - LocalDate

```
LocalDate localDate = LocalDate.now();  
LocalDate.of(2015, 02, 20);  
LocalDate.parse("2015-02-20");
```

- Fechas y horas locales (sin TimeZone)
 - Manipulación de LocalDate

```
LocalDate tomorrow = LocalDate.now().plusDays(1);
```

```
LocalDate previousMonthSameDay =  
    LocalDate.now().minus(1, ChronoUnit.MONTHS);
```

```
LocalDate firstDayOfMonth = date.  
    .with(TemporalAdjusters.firstDayOfMonth());
```

- Fechas y horas locales (sin TimeZone)
 - Acceso a las partes de la LocalDate

```
DayOfWeek sunday =  
    LocalDate.parse("2016-06-12").getDayOfWeek();  
  
int twelve =  
    LocalDate.parse("2016-06-12").getDayOfMonth();
```

- Fechas y horas locales (sin TimeZone)
 - Comparaciones de LocalDate

```
boolean notBefore = LocalDate.now().isBefore(date);
```

```
boolean notBefore = LocalDate.now().isAfter(date);
```

- Fechas y horas locales (sin TimeZone)
 - `LocalTime` (Horas, minutos y segundos)

```
LocalTime now = LocalTime.now();
```

```
LocalTime sixThirty = LocalTime.parse("06:30");
```

```
LocalTime sixThirty = LocalTime.of(6, 30);
```

- Fechas y horas locales (sin TimeZone)
 - Manipulación de LocalTime

```
LocalTime sevenThirty =  
    time.plus(1, ChronoUnit.HOURS);
```

- Acceso a las partes de la hora

```
int six = LocalTime.parse("06:30").getHour();
```

- Fechas y horas locales (sin TimeZone)
 - LocalDateTime

```
LocalDateTime.now();
```

```
LocalDateTime.parse("2015-02-20T06:30:00");
```

```
LocalDateTime.of(2015, Month.FEBRUARY, 20, 06, 30);
```


- Fechas y horas locales (sin TimeZone)
 - Manipulación de LocalDateTime

```
localDateTime.plusDays(1);
```

```
localDateTime.minusHours(2);
```

- Acceso a las partes del LocalDateTime

```
localDateTime.getMonth();
```

- Fechas y horas con una zona horaria (time zone) específica
 - Zonas horarias con ZoneId

```
ZoneId zoneId = ZoneId.of("Europe/Paris");  
  
Set<String> ids = ZoneId.getAvailableZoneIds();
```

- Fechas y horas con una zona horaria (time zone) específica
 - ZonedDateTime

```
ZoneId zoneId = ...
```

```
ZonedDateTime dateTime =  
    ZonedDateTime.of(localDateTime, zoneId);
```

```
ZonedDateTime.parse("2015-05-03T10:15:30+01:00[Europe/Paris]");
```

- Fechas y horas con un desplazamiento (offset)
 - ZoneOffset y OffsetDateTime

```
LocalDateTime localDateTime = ...  
  
ZoneOffset offset = ZoneOffset.of("+02:00");  
  
OffsetDateTime offSetByTwo = OffsetDateTime  
    .of(localDateTime, offset);
```

- Intervalos temporales
 - Duration: Segundos y nanosegundos
 - Period: Años, meses y días

```
LocalDate initialDate = ...  
LocalDate finalDate = ...  
  
int days = Period.between(finalDate, initialDate).getDays();  
  
int days =  
    ChronoUnit.DAYS.between(finalDate , initialDate);
```

- Intervalos temporales
 - Duration: Segundos y nanosegundos
 - Period: Años, meses y días

```
LocalTime initialTime = ...  
LocalTime finalTime = ...  
  
int sec =  
    Duration.between(finalTime, initialTime).getSeconds();  
  
int sec =  
    ChronoUnit.SECONDS.between(finalTime, initialTime);
```

Ejercicio 4

- Implementa un programa que cuente cuántas veces tu cumpleaños será en cada uno de los días de la semana en los próximos 40 años
- Implementa otro programa que te indique cuántos días, horas, minutos y segundos faltan para tu próximo cumpleaños



9 - 13

- **Módulos Java**
 - Proyecto Jigsaw
 - Java Platform Module System (JPMS)
 - No se ha extendido mucho su uso
- **Métodos privados en interfaces**
 - Para reutilización de implementación en los métodos por defecto
- **Try-With-Resources con variables existentes**
 - Para mejorar la legibilidad

- Métodos para crear colecciones
- Mejoras en los streams
- Cliente HTTP2
- API para gestionar procesos del SO
- Reactive Streams (más eficientes)

- **Lenguaje**
 - Inferencia de tipos en variables locales
- **API**
 - Colecciones no modificables
 - Mejoras en la clase Optional

- **Lenguaje**
 - Usar var para declarar parámetros en las expresiones lambda
- **API**
 - Mejoras en String
 - Mejoras en Path
 - Mejoras en Optional
 - Mejoras en entrada/salida
 - Mejoras en las colecciones

- **Lenguaje**
 - Sentencia switch como expresiones
- **API**
 - Mejoras en String
 - Mejoras en los Collectors de Stream
 - Mejoras en manejo de ficheros

- Lenguaje
 - Mejoras en las expresiones switch
 - Texto en bloques