

Curso JAVA 8

Enero 2018

José María González
consultas@avante.es

www.avante.es avante@avante.es 902 117 902

avante
¿hasta dónde quieras llegar?



PRESENTACIÓN

- Sobre el curso
- Presentación
- Alcance
- Metodología
- Descansos

ÍNDICE

- Tema 0 Presentación de JAVA y versiones.
- Tema 1 Repaso y Mejoras en el lenguaje
- Tema 2 Repaso y Cambios en Clases e Interfaces
- Tema 3 Lambdas
- Tema 4 Streams
- Tema 5 Cambios en APIs
- Tema 6 Time



TEMA 0: PRESENTACIÓN DE JAVA Y VERSIONES

Características de JAVA

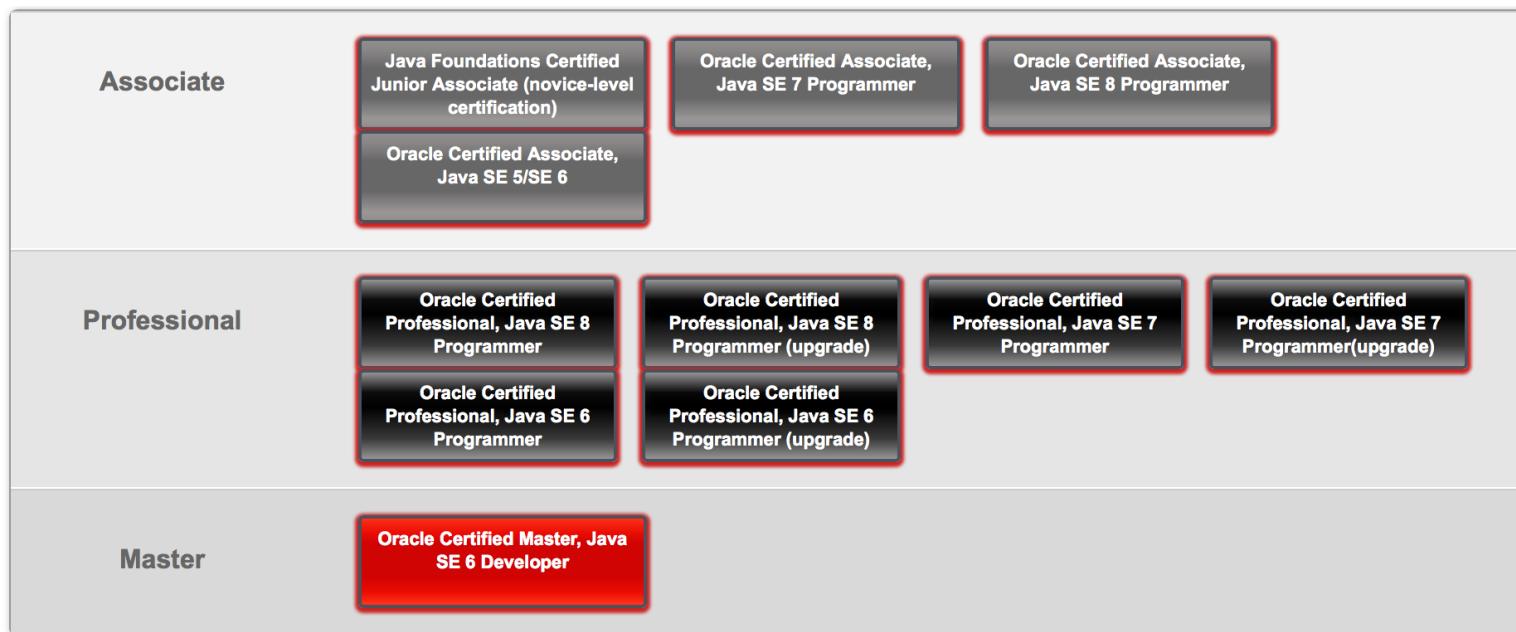
- Orientado a objetos
- Permite encapsulación
- Independiente de la plataforma “Write once, run everywhere.”
- Robusto gracias a su gestión de la memoria
- Simple
- Seguro al ejecutarse en una máquina virtual que crea un sandbox de ejecución.
- Tipos de ediciones de Java
 - JRE
 - JSE
 - JEE



TEMA 0: PRESENTACIÓN DE JAVA Y VERSIONES

Certificaciones de JAVA

- Presentación del learning path de JAVA
- Associate level certification (Java SE)
- Professional level certification (Java SE)



TEMA 0: PRESENTACIÓN DE JAVA Y VERSIONES

JAVA 5.0 a JAVA 6.0

- Mejoras en colecciones
- Mejoras de rendimiento
- Mejoras en Swing
- Mejoras en la instrumentación y gestión de la máquina virtual
- Incluye un sistema para ejecutar scripts (como Javascript)

TEMA 0: PRESENTACIÓN DE JAVA Y VERSIONES

JAVA 6.0 a JAVA 7.0

- Mejoras en Swing
- Mejoras en gestión de ficheros
- Mejoras de seguridad (por ejemplo acceso al keychain de Apple)
- Creación de la gestión nativa de XML en Java a través JAXB, JAX-WS y JAXP



TEMA 0: PRESENTACIÓN DE JAVA Y VERSIONES

JAVA 7.0 a JAVA 8.0

- Se incorporan elementos de programación funcional como lambdas y streams.
- Mejoras en seguridad (nuevas maneras de encriptar y más certificados)
- Mejoras en JavaFX
- Mejoras en el manejo de fechas

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

Vamos a repasar conceptos del lenguaje Java con nuevas características y mejoras que se han producido en la versión 8

- Vamos a revisar conceptos de Java a nivel de certificación para evitar posibles errores en la migración a Java 8 y reforzar conocimientos.
- Explicaremos nuevos conceptos del lenguaje y los trabajaremos con ejercicios.

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Tipos primitivos

Keyword	Type	Example
boolean	true or false	true
byte	8-bit integral value	123
short	16-bit integral value	123
int	32-bit integral value	123
long	64-bit integral value	123
float	32-bit floating-point value	123.45f
double	64-bit floating-point value	123.456
char	16-bit Unicode value	'a'

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Un short va de -128 a 127.
- Octales binarios y hexadecimales

```
System.out.println(56);           // 56
System.out.println(0b11);          // 3
System.out.println(017);           // 15
System.out.println(0x1F);          // 31
```

- Java permite usar _ para visualizar mejor las cifras

```
int million1 = 1000000;
int million2 = 1_000_000;
```

```
double notAtStart = _1000.00;      // DOES NOT COMPILE
double notAtEnd = 1000.00_;         // DOES NOT COMPILE
double notByDecimal = 1000_.00;      // DOES NOT COMPILE
double annoyingButLegal = 1_00_0.0_0; // this one compiles
```

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Declaración múltiple de variables: ¡Siempre del mismo tipo! Se le asigna un valor a cada una por separado.
- Identificadores: Solo hay tres reglas
 - Empiezan por una letra, \$ o _
 - Después pueden contener números
 - No se pueden usar palabras reservadas

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

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



TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Inicialización de variables locales. Vigilar si no compilan por no estar inicializadas y vigilar el scope.
- Inicialización de variables de instancia y de clase (o estáticas).

Variable type	Default initialization value
<code>boolean</code>	<code>false</code>
<code>byte, short, int, long</code>	<code>0</code> (in the type's bit-length)
<code>float, double</code>	<code>0.0</code> (in the type's bit-length)
<code>char</code>	<code>'\u0000'</code> (NUL)
All object references (everything else)	<code>null</code>

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Orden de elementos en las clases

Element	Example	Required?	Where does it go?
Package declaration	<code>package abc;</code>	No	First line in the file
Import statements	<code>import java.util.*;</code>	No	Immediately after the package
Class declaration	<code>public class C</code>	Yes	Immediately after the import
Field declarations	<code>int value;</code>	No	Anywhere inside a class
Method declarations	<code>void method()</code>	No	Anywhere inside a class

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- El colector de basura.
 - Cuando el objeto no tiene referencias apuntándole
 - Cuando todas las referencias están fuera del scope de ejecución.
 - System.gc() pide a la JVM que borre los elementos en memoria que no se usan. No se garantiza que se ejecute.
 - protected void finalize(){} se ejecuta si se ejecuta el garbage collector. Si el gc falla por cualquier motivo no se ejecuta de nuevo

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Operadores de Java y precedencia

Operator	Symbols and examples
Post-unary operators	<i>expression++ , expression--</i>
Pre-unary operators	<i>++ expression, --expression</i>
Other unary operators	<i>+, -, !</i>
Multiplication/Division/Modulus	<i>* , / , %</i>
Addition/Subtraction	<i>+ , -</i>
Shift operators	<i><<, >>, >>></i>
Relational operators	<i>< , > , <= , >= , instanceof</i>
Equal to/not equal to	<i>== , !=</i>
Logical operators	<i>& , ^ , </i>
Short-circuit logical operators	<i>&& , </i>
Ternary operators	<i>boolean expression ? expression1 : expression2</i>
Assignment operators	<i>= , += , -= , *= , /= , %= , &= , ^= , != , <= , >= , <<= , >>=</i>

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Operadores
 - Binarios: requieren dos operandos
 - Unarios: requieren solo uno
- Promoción de primitivas: las primitivas al operar entre ellas pueden ser promocionar a valores mayores. Estas son las reglas
 - Si dos valores tienen datos diferentes java promociona al mayor
 - Si uno es un entero y el otro un real java promociona el entero a real
 - Los valores pequeños (byte, short y char) se promocionan a int siempre que se use algún operador binario (+, *, -, ...) incluso si ninguno es int.
 - El valor resultante al que se le asigna debe tener el mismo valor que el promocionado.

```
short a = 1;  
short b = 2;  
short c = a + b;
```

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Operadores unarios

Unary operator	Description
+	Indicates a number is positive, although numbers are assumed to be positive in Java unless accompanied by a negative unary operator
-	Indicates a literal number is negative or negates an expression
++	Increments a value by 1
--	Decrement a value by 1
!	Inverts a Boolean's logical value

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Casting de primitivos. Cuidado con expresiones que no tienen casting y pueden promocionar

```
short x = 10;
short y = 3;
short z = x * y;
```

- Con los operadores compuestos también debemos vigilar

```
long x = 10;
int y = 5;
y = y * x; // DOES NOT COMPILE
```

```
long x = 10;
int y = 5;
y *= x;
```

- Operadores lógicos (también tenemos los de circuito corto && y ||)

$x \& y$
(AND)

	$y = true$	$y = false$
$x = true$	true	false
$x = false$	false	false

$x | y$
(INCLUSIVE OR)

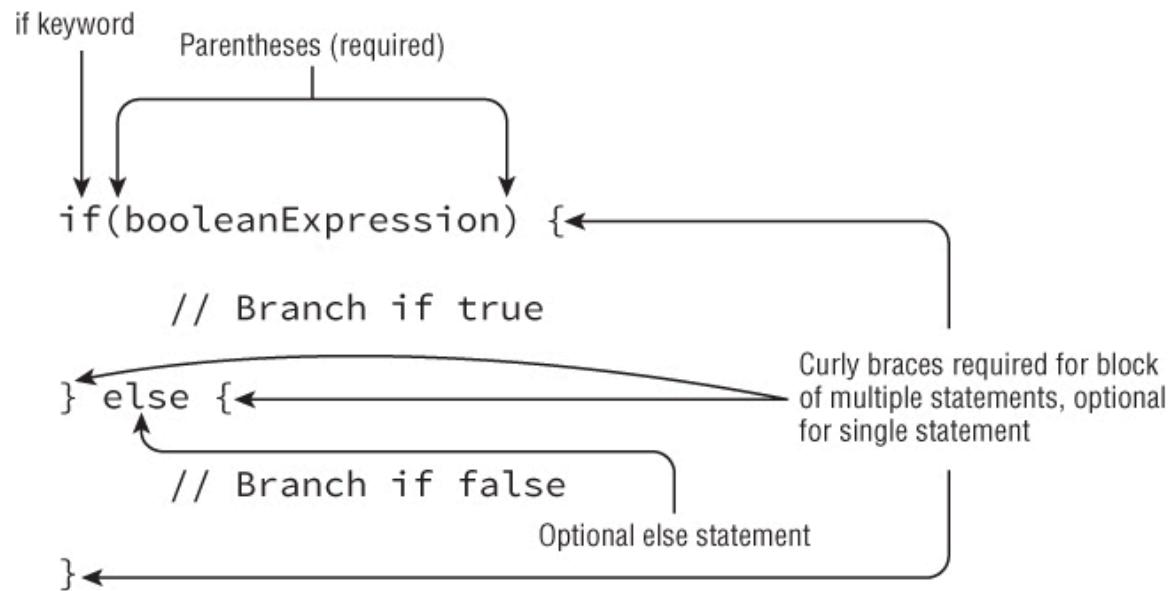
	$y = true$	$y = false$
$x = true$	true	true
$x = false$	true	false

$x ^ y$
(EXCLUSIVE OR)

	$y = true$	$y = false$
$x = true$	false	true
$x = false$	true	false

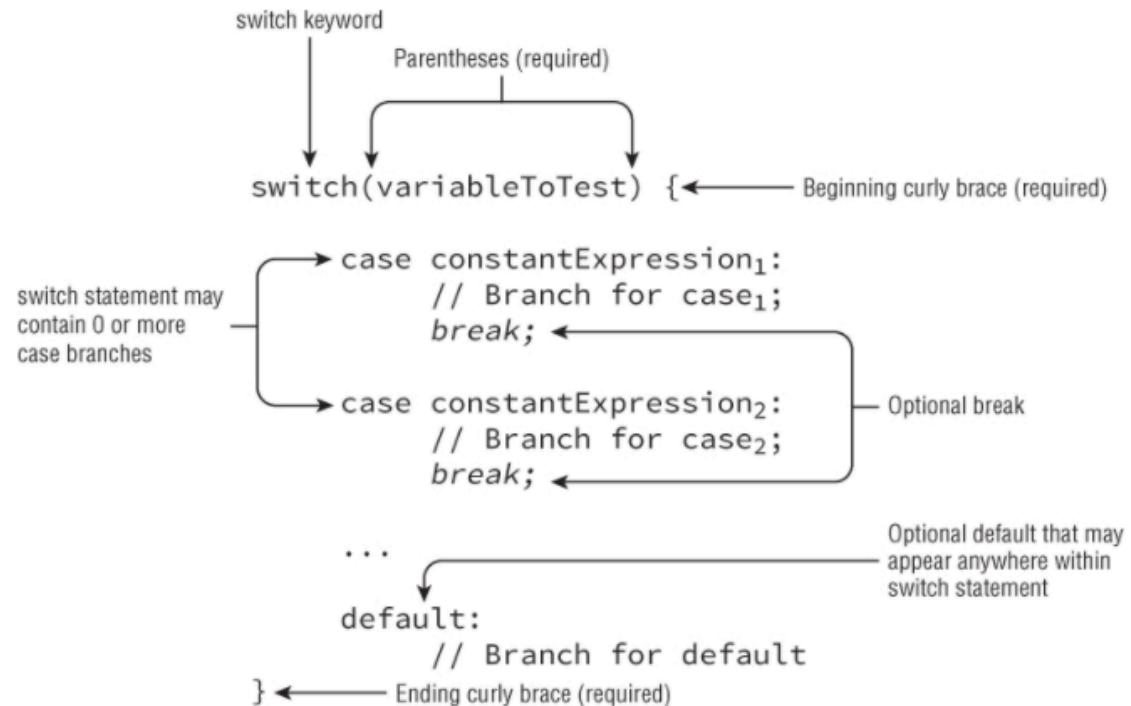
TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Operadores de igualdad (== y !=). Hay tres escenarios a contemplar
 - Hay promoción para tipos primitivos 4 == 4.0 es true ya que el 4 se promociona a double.
 - Comparación de booleanos
 - Comparación de objetos, incluidos null y valores de String
- If statement



TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Operador ternario booleanExpression ? expression1 : expression2 (solo evalúa la expresión que se ejecuta)
- Switch statement



TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Switch ahora soporta más tipos pero solo podemos usar valores resueltos en tiempo de compilación o finales para las comparaciones en los case.
 - `int` and `Integer`
 - `byte` and `Byte`
 - `short` and `Short`
 - `char` and `Character`
 - `int` and `Integer`
 - `String`
 - `enum` values

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

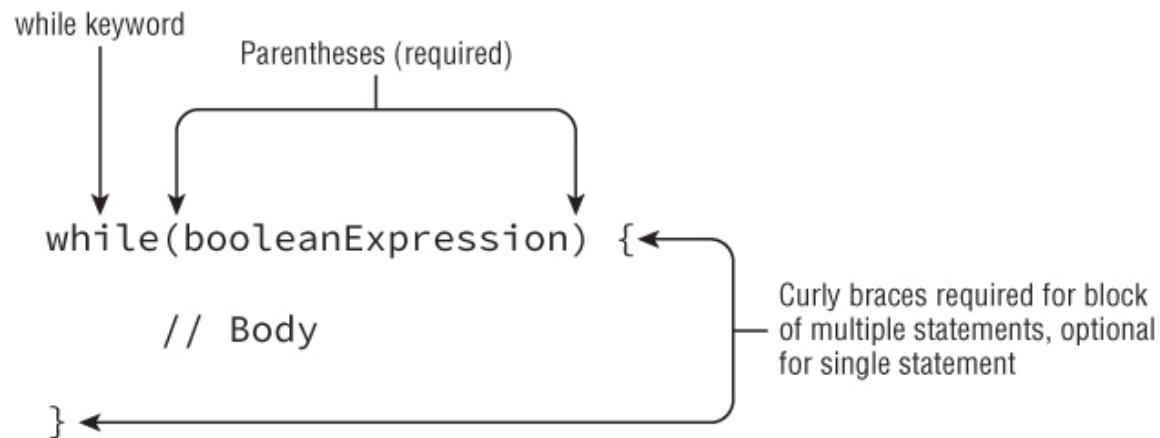
- Un ejemplo de switch

```
private int getSortOrder(String firstName, final String lastName) {  
    String middleName = "Patricia";  
    final String suffix = "JR";  
    int id = 0;  
    switch(firstName) {  
        case "Test":  
            return 52;  
        case middleName:      // DOES NOT COMPILE  
            id = 5;  
            break;  
        case suffix:  
            id = 0;  
            break;  
        case lastName: // DOES NOT COMPILE  
            id = 8;  
            break;  
        case 5: // DOES NOT COMPILE  
            id = 7;  
            break;  
        case 'J': // DOES NOT COMPILE  
            id = 10;  
            break;  
        case java.time.DayOfWeek.SUNDAY: // DOES NOT COMPILE  
            id=15;  
            break;  
    }  
    return id;  
}
```

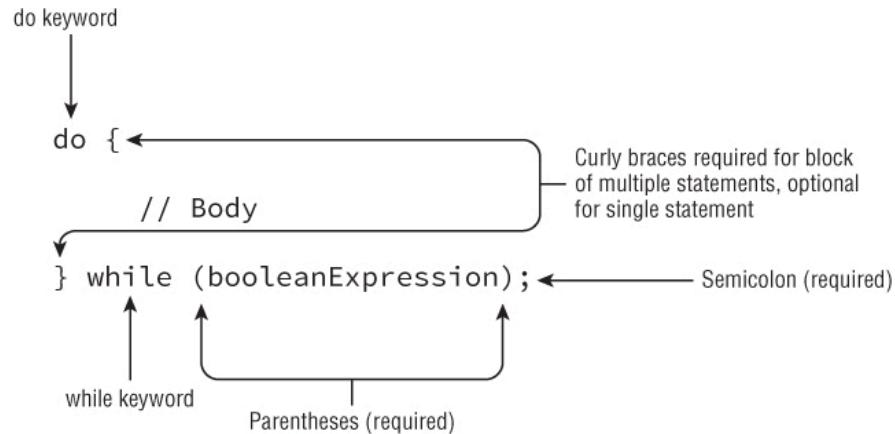


TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- While statement

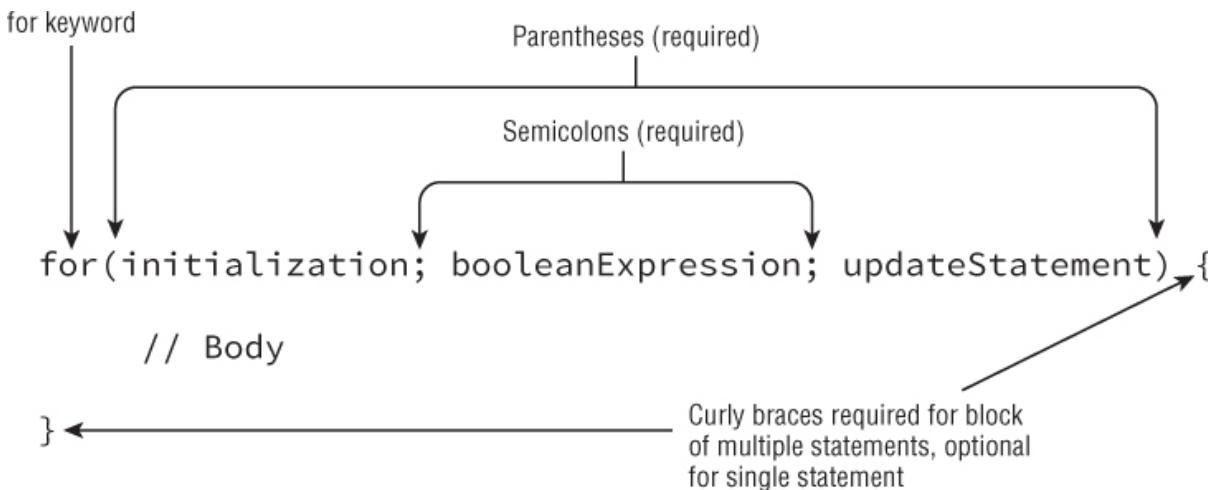


- do-while statement



TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

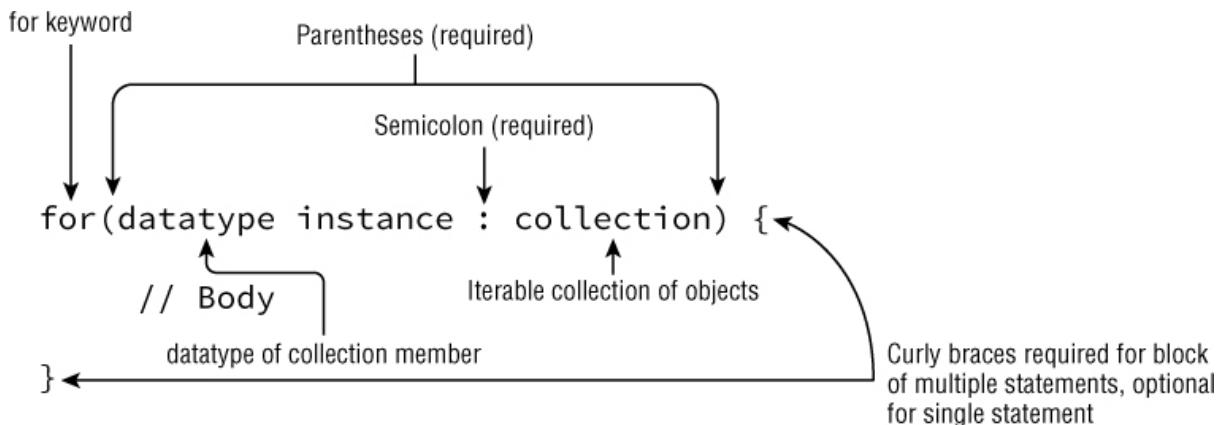
- **for statement**



- ① Initialization statement executes
- ② If booleanExpression is true continue, else exit loop
- ③ Body executes
- ④ Execute updateStatements
- ⑤ Return to Step 2

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Debemos tener varias cosas en cuenta para un bucle for
 - Se pueden crear bucles infinitos for(; ;)
 - Se pueden añadir múltiples términos. for(long y = 0, z = 4; x < 5 && y < 10; x++, y++)
 - Debemos tener cuidado de no redeclarar variables en la inicialización.
 - Hay que vigilar tipos incompatibles en la inicialización.
 - Hay que vigilar no usar variables declaradas en el for fuera del mismo.
- for each statement. Es una adaptación del for para arrays y objetos que implementen la interfaz Collection

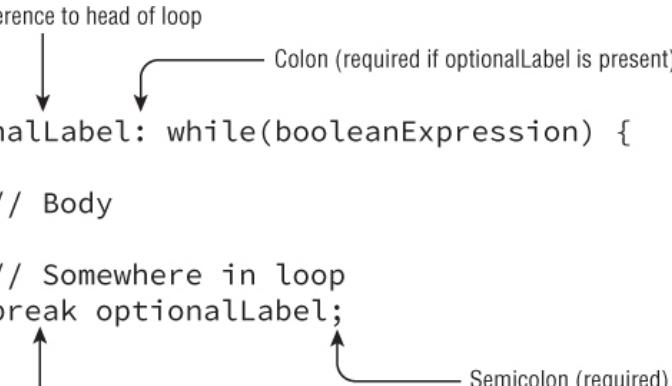


TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Para trabajar con colecciones en el for each solo tenemos que conocer List y ArrayList.
- Es importante estar familiarizado con los bucles anidados con varios elementos
 - break
 - continue
 - se pueden crear etiquetas para los bucles que pueden ser referenciadas por break y continue

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

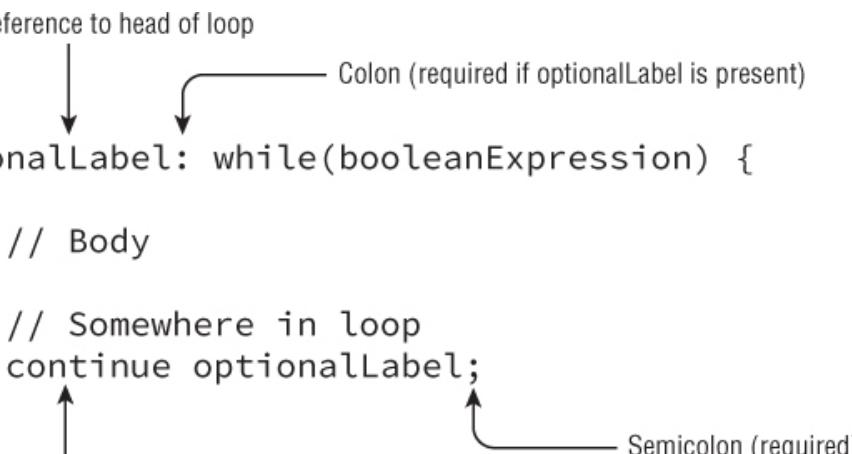
Optional reference to head of loop



```
optionalLabel: while(booleanExpression) {
    // Body
    // Somewhere in loop
    break optionalLabel;
}
```

The diagram shows a code snippet for a while loop. An arrow points from the label 'optionalLabel' to the start of the loop. Another arrow points from the colon after 'optionalLabel' to the text 'Colon (required if optionalLabel is present)'. A third arrow points from the 'break' keyword to the label 'break keyword'. A fourth arrow points from the semicolon at the end of the loop body to the text 'Semicolon (required)'.

Optional reference to head of loop



```
optionalLabel: while(booleanExpression) {
    // Body
    // Somewhere in loop
    continue optionalLabel;
}
```

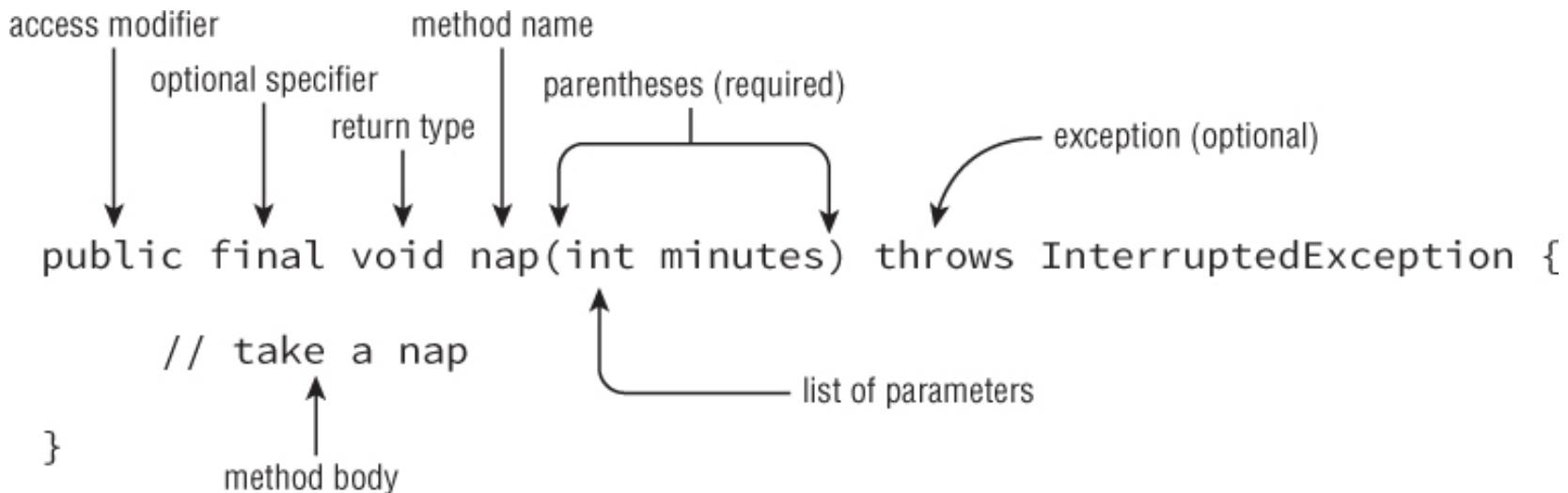
The diagram shows a code snippet for a while loop. An arrow points from the label 'optionalLabel' to the start of the loop. Another arrow points from the colon after 'optionalLabel' to the text 'Colon (required if optionalLabel is present)'. A third arrow points from the 'continue' keyword to the label 'continue keyword'. A fourth arrow points from the semicolon at the end of the loop body to the text 'Semicolon (required)'.

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

	Allows optional labels	Allows <i>break</i> statement	Allows <i>continue</i> statement
if	Yes *	No	No
while	Yes	Yes	Yes
do while	Yes	Yes	Yes
for	Yes	Yes	Yes
switch	Yes	Yes	No

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Elementos de un método



TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

Element	Value in nap() example	Required?
Access modifier	public	No
Optional specifier	final	No
Return type	void	Yes
Method name	nap	Yes
Parameter list	(int minutes)	Yes, but can be empty parentheses
Optional exception list	throws InterruptedException	No
Method body	{ // take a nap }	Yes, but can be empty braces



TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Java tiene 4 modificadores de acceso:
 - public: el método puede ser llamado desde cualquier clase
 - private: el método solo se puede llamar desde la misma clase
 - protected: el método solo se puede llamar desde clases del mismo paquete o subclases
 - default (sin modificador): el método solo se puede llamar desde clases del mismo paquete
- Otros especificadores opcionales
 - static para métodos de clases
 - abstract Usado cuando no se provee un cuerpo del método
 - final usado cuando no se puede extender en subclases
 - synchronized para hacer segura la ejecución de un método en diferentes hilos.
 - native Para ejecutar código en otros idiomas como C.
 - Strictfp Para cálculos estrictos de la coma flotante.

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Es muy importante el orden de los especificadores. Sobre todo tener en cuenta que el tipo de retorno debe ir junto al nombre del método y es obligatorio.
- El nombre del método sigue la misma convención que los nombres de clases aunque suelen empezar por minúscula por convención
- Los parámetros aparecen separados por coma así como la lista de excepciones.
- Los varargs deben estar al final y solo puede haber uno.

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Revisemos los modificadores de acceso

Can access	If that member is private?	If that member has default (package private) access?	If that member is protected?	If that member is public?
Member in the same class	Yes	Yes	Yes	Yes
Member in another class in same package	No	Yes	Yes	Yes
Member in a superclass in a different package	No	No	Yes	Yes
Method/field in a non-superclass class in a different package	No	No	No	Yes

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Los métodos estáticos no requieren una instancia para utilizarlos. Es importante vigilar que antes de dar una excepción por nulo que la variable no sea estática.
- Hay que vigilar también que los métodos estáticos no pueden llamar a métodos no estáticos sin una instancia de por medio
- Debemos tener esta tabla clara y validar siempre cuando haya métodos estáticos

Type	Calling	Legal?	How?
Static method	Another static method or variable	Yes	Using the classname
Static method	An instance method or variable	No	
Instance method	A static method or variable	Yes	Using the classname or a reference variable
Instance method	Another instance method or variable	Yes	Using a reference variable

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Para declarar constantes podemos usar esta nomenclatura private static final int NUM_BUCKETS = 45;
- Podemos usar import estáticos para hacer referencias a métodos para usarlos sin el identificador de clase. Por ejemplo import static java.util.Arrays.asList; o import static java.util.Arrays.*; Para los estáticos sólo podemos referenciar métodos, no podemos hacer referencia a la clase o al * de paquete.
- Java pasa parámetros por valor. Se hace una copia del valor y se pasa. Con primitivas y objetos inmutables nunca cambia el valor.

```
public static void main(String[] args) {  
    int num = 4;  
    newNumber(5);  
    System.out.println(num);      // 4  
}  
public static void newNumber(int num) {  
    num = 8;  
}
```

```
public static void main(String[] args) {  
    String name = "Webby";  
    speak(name);  
    System.out.println(name);  
}  
public static void speak(String name) {  
    name = "Sparky";  
}
```

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Con una referencia cambia un poco la cosa
 - public static void main (String[] args){
 - StringBuilder name = new StringBuilder("Webby");
 - speak (name);
 - System.out.println(name);
 - }
 - public static void speak (StringBuilder name){
 - name.append ("Sparky");
 - }
- El resultado sería WebbySparky



TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- La sobreescritura de métodos la llevamos viendo desde el principio del curso. Hay algunas reglas que conviene recordar
 - Hay que vigilar arrays y varargs ya que Java trata los varargs como un array
 - Los parámetros pueden variar
 - Java no compila con los mismos parámetros y distinto tipo retornado
 - Tampoco es válido si tienen los mismos parámetros y uno es estático y el otro no.
 - Al hacer autoboxing java es capaz de determinar si es un primitivo o un wrapper o si encaja en una clase de la que herede.
 - Con las primitivas también se hace promoción cuando sea necesario
 - Un pequeño resumen

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

Rule	Example of what will be chosen for <code>glide(1,2)</code>
Exact match by type	<code>public String glide(int i, int j) {}</code>
Larger primitive type	<code>public String glide(long i, long j) {}</code>
Autoboxed type	<code>public String glide(Integer i, Integer j) {}</code>
Varargs	<code>public String glide(int... nums) {}</code>



TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Los constructores deben llamarse igual que la clase y no devuelven tipo alguno. Se instancian con new. Debemos vigilar que no intenten pasar un método como constructor y o bien devuelva algo o no tenga exactamente el nombre de la clase.
- Es importante recordar que por defecto se llama a super()
- Los constructores tienen modificadores de acceso. ¿Para qué podríamos necesitar un constructor privado?
- También se pueden sobrecargar los constructores con reglas similares a los métodos.
- Con sobrecarga podemos usar this() para referenciar a otros constructores.
- El orden de inicializaciones se expande un poco
 - Si hay una super clase se inicializa primero (una llamada a super por ejemplo)
 - Variables e inicializadores estáticos en el orden que aparezcan.
 - Variables e inicializadores de instancia en el orden que aparezcan.
 - El constructor.
- La encapsulación es un mecanismo de Java muy importante para el desarrollo y una de las características principales de Java. Hay unas convenciones de nombres a seguir



TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

Rule	Example
Properties are private.	<pre>private int numEggs;</pre>
Getter methods begin with <code>is</code> if the property is a boolean.	<pre>public boolean isHappy() { return happy; }</pre>
Getter methods begin with <code>get</code> if the property is not a boolean.	<pre>public int getNumEggs() { return numEggs; }</pre>
Setter methods begin with <code>set</code> .	<pre>public void setHappy(boolean happy) { this.happy = happy; }</pre>
The method name must have a prefix of <code>set/get/is</code> , followed by the first letter of the property in uppercase, followed by the rest of the property name.	<pre>public void setNumEggs(int num) { numEggs = num; }</pre>

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Hemos hablado de clases inmutables. Para crearlas simplemente no generamos setters para que solo se puedan definir en la construcción.
- Los tipos enumerados nos dan cobertura para un conjunto de constantes

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

- Los tipos enumerados son muy útiles para su uso con Switch o para modelar atributos con valores fijos.
- Los tipos enumerados son muy potentes en Java y permiten modelar no solo un listado de atributos pero también asociarlo a valores.

TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- Veamos un ejemplo más complejo

```
package com.avante.util;

public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS (4.869e+24, 6.0518e6),
    EARTH (5.976e+24, 6.37814e6),
    MARS (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27, 7.1492e7),
    SATURN (5.688e+26, 6.0268e7),
    URANUS (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);

    private final double mass; // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    @SuppressWarnings("unused")
    private double mass() { return mass; }
    @SuppressWarnings("unused")
    private double radius() { return radius; }

    // universal gravitational constant (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
    public static void main(String[] args) {
        double earthWeight = 70;
        double mass = earthWeight/EARTH.surfaceGravity();
        for (Planet p : Planet.values())
            System.out.printf("Your weight on %s is %F%n",
                p, p.surfaceWeight(mass));
    }
}
```

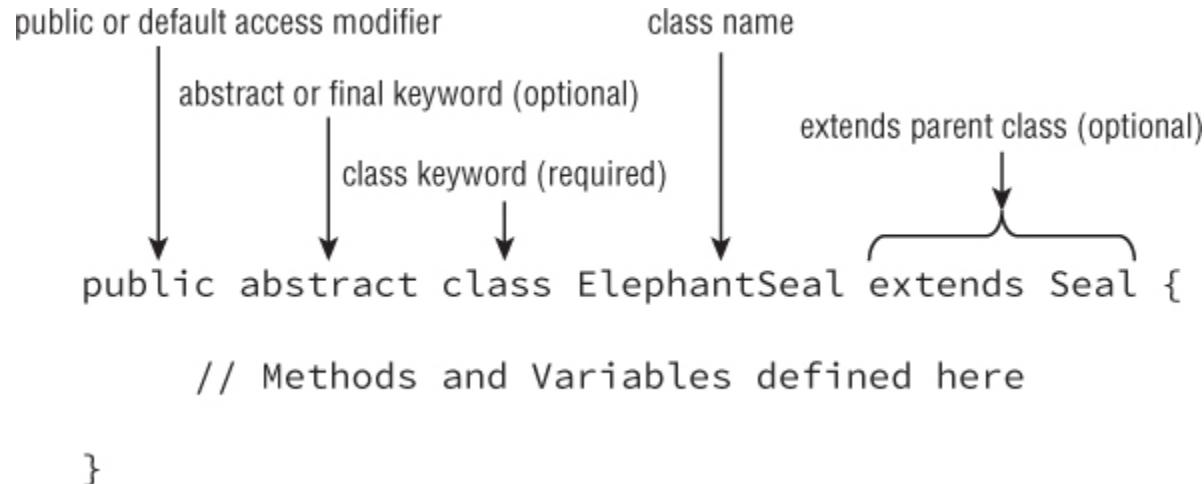
TEMA 1: REPASO Y MEJORAS EN EL LENGUAJE

- En Java podemos indicar el tipo de una lista u otro objeto con el operador de diamante <>. Desde Java 7 podemos no declarar la parte de la derecha

```
List<String> list1 = new ArrayList<String>();  
List<String> list2 = new ArrayList<>();
```

TEMA 2: REPASO Y MEJORAS EN INTERFACES

- Java permite que unas clases hereden de otras. Pero no permite la herencia múltiple. Todas las clases extienden de Object.



The diagram illustrates the structure of a Java class definition. It shows the following components and their relationships:

- public or default access modifier**: Points to the word "public".
- abstract or final keyword (optional)**: Points to the word "abstract".
- class keyword (required)**: Points to the word "class".
- class name**: Points to the identifier "ElephantSeal".
- extends parent class (optional)**: Points to the word "extends". A brace groups "extends Seal" together.

Below the class definition, the text "// Methods and Variables defined here" is followed by a closing brace "}" on a new line.

```
public abstract class ElephantSeal extends Seal {  
    // Methods and Variables defined here  
}
```

TEMA 2: REPASO Y MEJORAS EN INTERFACES

- Al extender una clase permitimos acceder a sus métodos (dependiendo de los modificadores) y hacerlos tuyos.
- Podemos acceder a constructores de la clase padre con super() (debe ser la primera línea dentro de un constructor) la llamada super() es implícita a todos los constructores.
- Veamos unas reglas
 - La primera llamada en un constructor debe ser a otro constructor usando la llamada this() o al constructor padre con super()
 - super() no se puede usar después de la primera llamada.
 - Si no hay ningún super() en el constructor Java meterá un super() sin argumentos de manera implícita.
 - Si el padre no tiene super() sin argumentos fallará si no se definen otros constructores.
 - Si el padre no tiene un constructor sin argumentos los constructores hijos tendrán que hacer una llamada explícita a otro super() con argumentos.

TEMA 2: REPASO Y MEJORAS EN INTERFACES

- Hay que vigilar los modificadores de acceso a la hora de usar los métodos del padre y vigilar los paquetes en los que se declaran.
- Podemos acceder a los atributos y métodos del padre a través de super.miAtributo o super.miMetodo()
- Para hacer la sobreescritura hay algunas reglas:
 - Debe tener la misma estructura que el padre
 - Tiene que tener el mismo o mayor acceso que el padre
 - No puede lanzar una excepción chequeada más amplia que el padre o nueva.
 - Si el hijo devuelve algún valor debe ser del mismo tipo o una subclase del mismo.

TEMA 2: REPASO Y MEJORAS EN INTERFACES

- Los métodos privados no se sobreesciben.
- Una clase hija puede ocultar de manera similar a la sobreescritura un método de la clase padre cuando ambos son estáticos. Para ello debe seguir estas reglas:
 - Debe tener la misma estructura que el padre
 - Tiene que tener el mismo o mayor acceso que el padre
 - No puede lanzar una excepción chequeada más amplia que el padre o nueva.
 - Si el hijo devuelve algún valor debe ser del mismo tipo o una subclase del mismo.
 - El método debe ser static al igual que el padre.

TEMA 2: REPASO Y MEJORAS EN INTERFACES

- La distinción entre sobreescribir y ocultar es que el sobreescribir reemplaza la clase padre la ocultación solo lo hace en el hijo. En los métodos sobreescritos el padre solo se llama de manera explícita. En el caso del oculto el padre siempre se ejecuta si se hace una llamada.
Veamos dos ejemplos:

```
public class Marsupial {  
    public static boolean isBiped() {  
        return false;  
    }  
    public void getMarsupialDescription() {  
        System.out.println("Marsupial walks on two legs: "+isBiped());  
    }  
}  
public class Kangaroo extends Marsupial {  
    public static boolean isBiped() {  
        return true;  
    }  
    public void getKangarooDescription() {  
        System.out.println("Kangaroo hops on two legs: "+isBiped());  
    }  
    public static void main(String[] args) {  
        Kangaroo joey = new Kangaroo();  
        joey.getMarsupialDescription();  
        joey.getKangarooDescription();  
    }  
}
```

Marsupial walks on two legs: false
Kangaroo hops on two legs: true

```
class Marsupial {  
    public boolean isBiped() {  
        return false;  
    }  
    public void getMarsupialDescription() {  
        System.out.println("Marsupial walks on two legs: "+isBiped());  
    }  
}  
public class Kangaroo extends Marsupial {  
    public boolean isBiped() {  
        return true;  
    }  
    public void getKangarooDescription() {  
        System.out.println("Kangaroo hops on two legs: "+isBiped());  
    }  
    public static void main(String[] args) {  
        Kangaroo joey = new Kangaroo();  
        joey.getMarsupialDescription();  
        joey.getKangarooDescription();  
    }  
}
```

Marsupial walks on two legs: true
Kangaroo hops on two legs: true

TEMA 2: REPASO Y MEJORAS EN INTERFACES

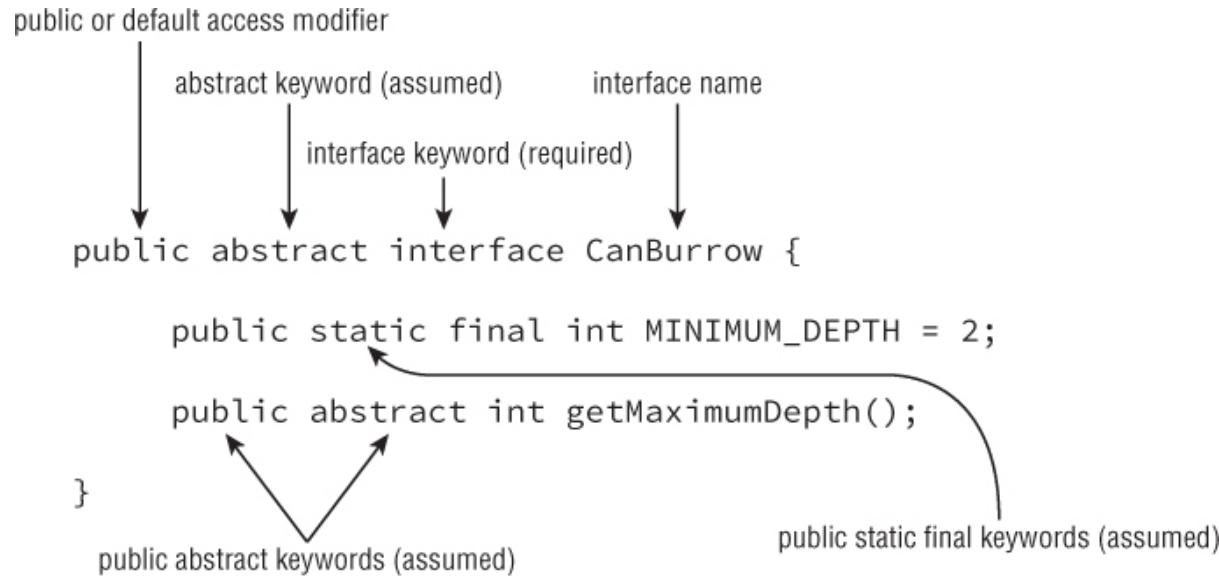
- Los métodos marcados con `final` no pueden ser sobreescritos.
- En cuanto a las variables solo se pueden ocultar, no se pueden sobreescribir.
- Las clases abstractas tienen la capacidad de implementar código que luego usarán las clases que las extiendan y a su vez definir métodos sin implementación que deben ser definidos por una clase hija
- Para definir una clase abstracta solo debemos indicar `abstract` antes de `class`.
- Los métodos sin implementación deben estar marcados como `abstract`. Obviamente no pueden ser privados ni finales.
- Las clases abstractas no se pueden instanciar, aunque si pueden ser referencias de clases hijas (al igual que los interfaces)

TEMA 2: REPASO Y MEJORAS EN INTERFACES

- Como reglas generales para las clases abstractas tendremos:
 - No se pueden instanciar directamente
 - Se pueden definir con cualquier número (incluso ninguno) de métodos abstractos o no.
 - no pueden ser marcadas como privadas o finales
 - las clases abstractas que extiendan otra clase abstractas heredan todos sus métodos y atributos pero no tienen que implementarlas si no quieren
 - La primera clase no abstracta debe implementar todos los métodos abstractos.
- Para los métodos abstractos debemos tener en cuenta lo siguiente:
 - Solo pueden ser definidos en clases abstractas
 - No pueden ser privados o finales
 - No pueden proveer ninguna implementación
 - Implementar un método abstracto sigue las mismas reglas que al extenderlo en cuanto a nombre, parámetro y visibilidad.

TEMA 2: REPASO Y MEJORAS EN INTERFACES

- Las interfaces nos ayudan a dar a una clase diversos comportamientos



TEMA 2: REPASO Y MEJORAS EN INTERFACES

- Estas son las reglas básicas de los interfaces:
 - No se pueden instanciar directamente.
 - No necesita declarar ningún método.
 - No puede ser marcado como final
 - Todos los interfaces por defecto son públicos y abstractos. Si no se especifica es implícito.
 - Todos los métodos, menos los default, por defecto son públicos y abstractos. Si no se especifica es implícito.
 - No extienden de Object como las clases y clases abstractas
- Una excepción importante de la herencia son los interfaces. Pueden extender más de una interfaz.
- No hay que implementar ningún método obviamente. La primera clase que implemente la interfaz debe implementar tanto los de ella como las interfaces padre.
- Las clases no pueden extender interfaces ni las interfaces extender clases
- No pasa nada si dos interfaces tienen el mismo método definido igual. Al implementarlo una clase cumpliría con ambos. La única excepción es si el tipo a devolver es distinto.

TEMA 2: REPASO Y MEJORAS EN INTERFACES

- Las variables en una interfaz son siempre públicas, estáticas y finales. El valor debe definirse en la declaración.
- Una novedad en Java 8 son los métodos default en los interfaces. Supone una pequeña variación en el concepto de los interfaces ya que permiten la implementación de código en los mismos.

```
public interface IsWarmBlooded {  
    boolean hasScales();  
    public default double getTemperature() {  
        return 10.0;  
    }  
}
```

- Estas son las reglas para métodos default
 - Sólo pueden declararse en interfaces. No en clases o clases abstractas.
 - Debe llevar la palabra default. Si la lleva debe implementar un body {}
 - No es necesario que sea estático o final y se puede sobreescibir por la clase que implemente la interfaz.
 - Debe ser público

TEMA 2: REPASO Y MEJORAS EN INTERFACES

Veamos un poco más en detalle los métodos default

- Al escribir una interfaz tenemos la posibilidad de generar métodos static que si tienen implementación. Normalmente estos métodos son utilizados como utilidades de nuestra clase y no tienen más uso que el de dar soporte a la clase que implementa un interfaz.
- Puede ocurrir que al evolucionar nuestra interfaz necesitemos adaptar el código antiguo en la interfaz para que siga siendo compatible con el nuevo. Por este motivo surgió la necesidad de tener métodos no estáticos en una interfaz.
- Permite sobrescribir el método a diferencia de los métodos estáticos que solo se ocultan.

TEMA 2: REPASO Y MEJORAS EN INTERFACES

- Si hay dos métodos estáticos iguales la clase que los implemente falla al compilar (no es igual que con los métodos normales)
- Java 8 también da cobertura para métodos estáticos dentro de los interfaces.
 - Serán públicos siempre.
 - Deben usar el nombre de la interfaz para referenciarlo.
- El polimorfismo permite a un objeto comportarse de manera diferente según la referencia que lo implemente. En ocasiones es muy útil dibujar bien la relación para entenderlo.
- El casting nos ayuda a encajar objetos en referencias siguiendo estas reglas
 - De una subclase a una clase padre no hace falta casting
 - De una superclase a una clase hija requiere un cast explícito.
 - Si no están relacionadas el compilador lanzará un error.
- En Java definimos un método virtual como aquel que no se sabe cual se ejecutará hasta que se ejecuta. En teoría todos son métodos virtuales menos los estáticos y/o finales y/o privados.

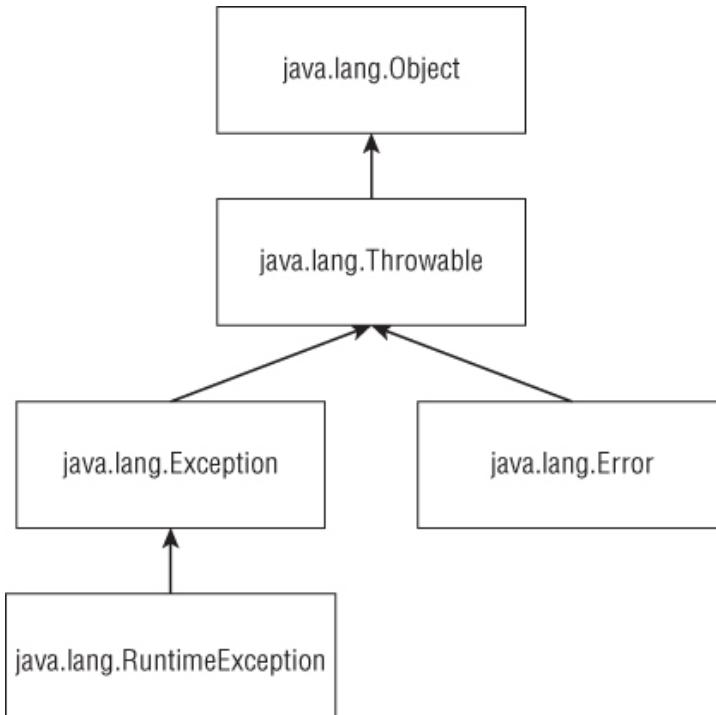
TEMA 2: REPASO Y MEJORAS EN INTERFACES

- Al igual que las referencias los parámetros de los métodos también pueden ser polimórficos.

```
public class Reptile {  
    public String getName() {  
        return "Reptile";  
    }  
}  
public class Alligator extends Reptile {  
    public String getName() {  
        return "Alligator";  
    }  
}  
public class Crocodile extends Reptile {  
    public String getName() {  
        return "Crocodile";  
    }  
}  
public class ZooWorker {  
    public static void feed(Reptile reptile) {  
        System.out.println("Feeding reptile "+reptile.getName());  
    }  
    public static void main(String[] args) {  
        feed(new Alligator());  
        feed(new Crocodile());  
        feed(new Reptile());  
    }  
}
```

TEMA 2B: EXCEPCIONES

- Las excepciones nos ayudan a controlar los posibles errores en nuestro código.
- Es fundamental tener el esquema de excepciones muy claro



TEMA 2B: EXCEPCIONES

- Debemos tener muy claro la siguiente regla. Si extiende de Exception pero no de RuntimeException es una excepción chequeada y debe tratarse o declararse siempre.
- Veamos todos los tipos de excepciones

Type	How to recognize	Okay for program to catch?	Is program required to handle or declare?
Runtime exception	Subclass of RuntimeException	Yes	No
Checked exception	Subclass of Exception but not subclass of RuntimeException	Yes	Yes
Error	Subclass of Error	No	No

TEMA 2B: EXCEPCIONES

- Las excepciones se usan de varias maneras, con un bloque try-catch-finally o con throws.
- Un try debe tener o un catch o un finally (Menos si es un try-with-resources).
- finally siempre debe ser el último en la lista. Finally siempre se ejecuta salvo que haya un System.exit que corta la ejecución del programa independientemente del finally.
- Puede haber varios catch para tratar diversas excepciones. Debemos tener en cuenta que deben ser de más específicos a más generales.

TEMA 2B: EXCEPCIONES

- Un ejemplo de excepciones es la posibilidad de lanzar una varias excepciones en cadena. Veamos un ejemplo

```
30: public String exceptions() {  
31:     String result = "*";  
32:     String v = null;  
33:     try {  
34:         try {  
35:             result += "before";  
36:             v.length();  
37:             result += "after";  
38:         } catch (NullPointerException e) {  
39:             result += "catch";  
40:             throw new RuntimeException();  
41:         } finally {  
42:             result += "finally";  
43:             throw new Exception();  
44:         }  
45:     } catch (Exception e) {  
46:         result += "done";  
47:     }  
48:     return result;  
49: }
```

- El resultado es *before catch finally done*. La línea 36 lanza un NullPointerException, ya no imprimirá la línea 37. La línea 38 recoge la excepción e imprime catch. Lanza una excepción que se recogerá más tarde pero primero imprime el finally. Al haber concluido el bloque interno se recoge la excepción en la línea 45 e imprime done.



TEMA 2B: EXCEPCIONES

- Debemos estar familiarizados con los tipos más comunes de excepciones.
 - Runtime Exceptions
 - ArithmeticException Thrown by the JVM when code attempts to divide by zero
 - ArrayIndexOutOfBoundsException Thrown by the JVM when code uses an illegal index to access an array
 - ClassCastException Thrown by the JVM when an attempt is made to cast an exception to a subclass of which it is not an instance
 - IllegalArgumentException Thrown by the programmer to indicate that a method has been passed an illegal or inappropriate argument
 - NullPointerException Thrown by the JVM when there is a null reference where an object is required
 - NumberFormatException Thrown by the programmer when an attempt is made to convert a string to a numeric type but the string doesn't have an appropriate format
 - Checked Exceptions
 - FileNotFoundException Thrown programmatically when code tries to reference a file that does not exist
 - IOException Thrown programmatically when there's a problem reading or writing a file
 - Errors
 - ExceptionInInitializerError Thrown by the JVM when a static initializer throws an exception and doesn't handle it
 - StackOverflowError Thrown by the JVM when a method calls itself too many times (this is called infinite recursion because the method typically calls itself without end)
 - NoClassDefFoundError Thrown by the JVM when a class that the code uses is available at compile time but not runtime



TEMA 2B: EXCEPCIONES

- Cuando un método puede lanzar una excepción chequeada tenemos dos opciones, declararla o gestionarla (handle)

```
public static void main(String[] args)
    throws NoMoreCarrotsException // declare exception
    eatCarrot();
}
public static void main(String[] args) {
    try {
        eatCarrot();
    } catch (NoMoreCarrotsException e) // handle exception
        System.out.print("sad rabbit");
    }
}
```

- Las excepciones como cualquier otra clase se pueden extender, aunque para ser considerada una excepción debe extender de Exception.
- Hay tres maneras de imprimir una excepción

```
5: public static void main(String[] args) {
6:     try {
7:         hop();
8:     } catch (Exception e) {
9:         System.out.println(e);
10:        System.out.println(e.getMessage());
11:        e.printStackTrace();
12:    }
13: }
14: private static void hop() {
15:     throw new RuntimeException("cannot hop");
16: }
```

TEMA 2B: EXCEPCIONES

- Es posible que tras utilizar un recurso java nos obligue a cerrarlo y en ocasiones podemos generar excepciones al intentar cerrar esos recursos. Un ejemplo típico puede ser la gestión de lectura de un fichero

```
BufferedReader br = null;
try {
    br = new BufferedReader(new FileReader("mifichero.txt"));
    System.out.println(br.readLine());
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (br != null)
            br.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

TEMA 2B: EXCEPCIONES

- Para eliminar este tratamiento final que no aporta nada realmente a nuestro código Java 7 incorpora try-with-resources

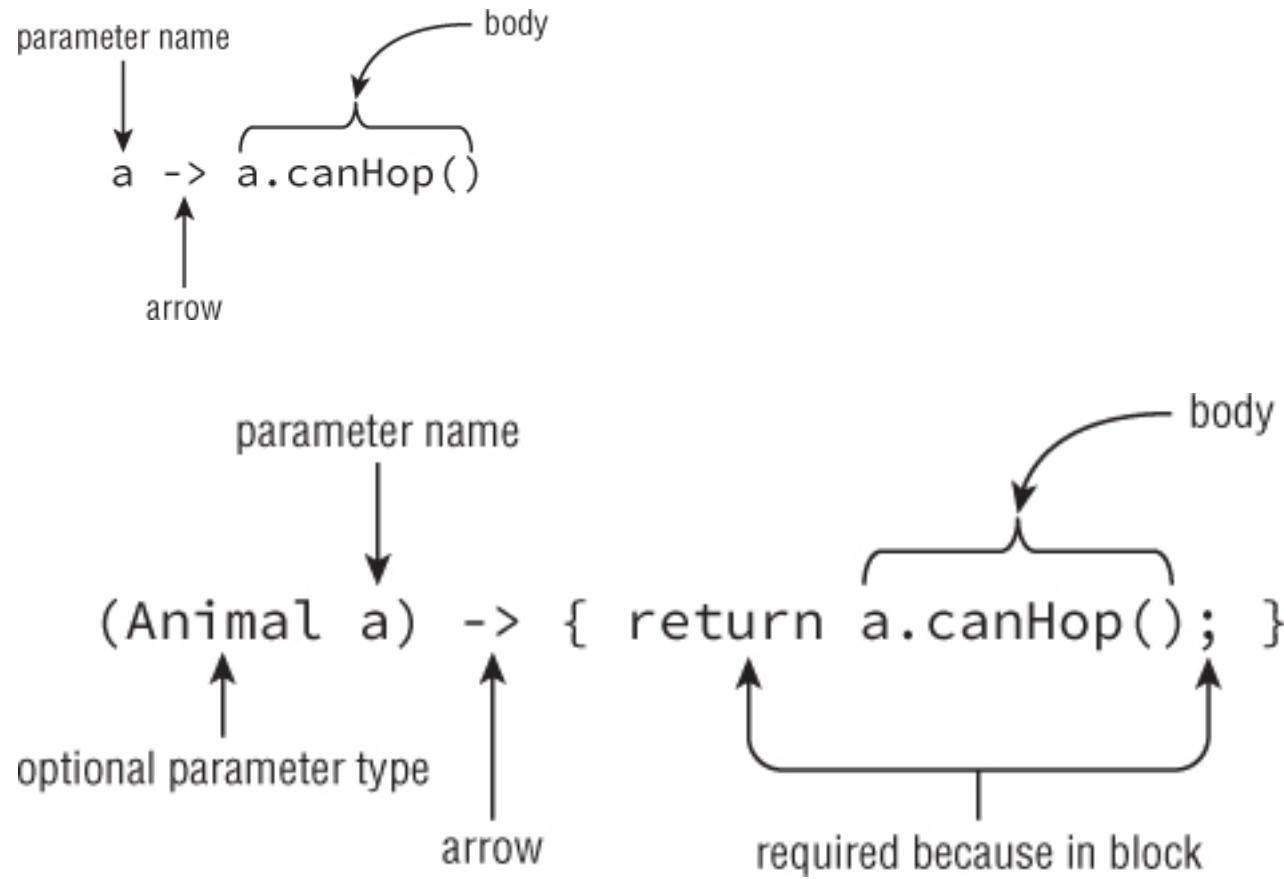
```
try (BufferedReader br2 = new BufferedReader(new FileReader("mifichero.txt"))) {  
    System.out.println(br2.readLine());  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

- Este recurso tiene implementada una nueva interfaz de Java llamada AutoCloseable que define que tiene que hacer para cerrar el recurso, por lo que evitamos tener que explicitarlo en nuestro código.

TEMA 3: LAMBDA

- Las lambdas son un añadido de la versión 8 que permite utilizar programación funcional en Java.
 - La idea de programación funcional es poder pasar a funciones bloques de código en lugar de tan solo variables o constantes. Esto permite reducir nuestro código evitándonos construir métodos repetitivos que solo tienen pequeñas diferencias.
 - Para usar lambdas usaremos la siguiente notación sencilla y completa

TEMA 3: LAMBdas



TEMA 3: LAMBDA

- Veamos algunos ejemplos de uso

```
3: print(() -> true);           // 0 parameters
4: print(a -> a.startsWith("test")); // 1 parameter
5: print((String a) -> a.startsWith("test")); // 1 parameter
6: print((a, b) -> a.startsWith("test")); // 2 parameters
7: print((String a, String b) -> a.startsWith("test")); // 2 parameters
```

- Veamos algunas mal formadas

```
print(a, b -> a.startsWith("test"));           // DOES NOT COMPILE
print(a -> { a.startsWith("test"); });          // DOES NOT COMPILE
print(a -> { return a.startsWith("test") });    // DOES NOT COMPILE
```

- si nos fijamos el primero no tiene paréntesis alrededor de las variables, el segundo no tienen return y el tercero le falta el punto y coma.

TEMA 3: LAMBDA

- Las lambdas tienen un método Predicate contra el que probar las expresiones funcionales. Básicamente el predicado nos permite chequear nuestra expresión.

```
1: import java.util.*;
2: import java.util.function.*;
3: public class PredicateSearch {
4:     public static void main(String[] args) {
5:         List<Animal> animals = new ArrayList<Animal>();
6:         animals.add(new Animal("fish", false, true));
7:
8:         print(animals, a -> a.canHop());
9:     }
10:    private static void print(List<Animal> animals, Predicate<Animal> checker) {
11:        for (Animal animal : animals) {
12:            if (checker.test(animal))
13:                System.out.print(animal + " ");
14:        }
15:        System.out.println();
16:    }
17: }
```



TEMA 3: LAMBDA

Veamos con más detalle las lambdas con un ejemplo

- Supongamos que estamos construyendo una red social e imaginemos que queremos habilitar al administrador a hacer cualquier acción sobre un grupo de usuarios.
- Si definimos una clase como esta

```
public class Person {  
  
    public enum Sex {  
        MALE, FEMALE  
    }  
  
    String name;  
    LocalDate birthday;  
    Sex gender;  
    String emailAddress;  
  
    public int getAge() {  
        // ...  
    }  
  
    public void printPerson() {  
        // ...  
    }  
}
```

TEMA 3: LAMBDA

- Si quisieramos crear filtros para varias casuísticas tendríamos que crear diversos métodos. Por ejemplo:

```
public static void printPersonsOlderThan(List<Person> roster, int age) {  
    for (Person p : roster) {  
        if (p.getAge() >= age) {  
            p.printPerson();  
        }  
    }  
}
```

- También podríamos crear métodos más genéricos

```
public static void printPersonsWithinAgeRange(  
    List<Person> roster, int low, int high) {  
    for (Person p : roster) {  
        if (low <= p.getAge() && p.getAge() < high) {  
            p.printPerson();  
        }  
    }  
}
```

TEMA 3: LAMBDA

- Una opción más compleja todavía es especificar el criterio de búsqueda en una clase local. Para ello crearemos una definición de las comprobaciones que queremos realizar

```
public static void printPersons(  
    List<Person> roster, CheckPerson tester) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            p.printPerson();  
        }  
    }  
}
```

- Una interfaz donde definimos lo que se puede testear

```
interface CheckPerson {  
    boolean test(Person p);  
}
```

TEMA 3: LAMBDA

- Una clase que implemente ese testeo podría ser la siguiente

```
class CheckPersonEligibleForSelectiveService implements CheckPerson {  
    public boolean test(Person p) {  
        return p.gender == Person.Sex.MALE &&  
            p.getAge() >= 18 &&  
            p.getAge() <= 25;  
    }  
}
```

- Y para hacer el testeo podemos hacer la llamada

```
printPersons(  
    roster, new CheckPersonEligibleForSelectiveService());
```



TEMA 3: LAMBDA

- La solución anterior aunque algo compleja nos permite no tener que reescribir código al cambiar los métodos ya que tenemos la interfaz.
- Otra solución podría ser el uso de clases anónimas

```
printPersons(  
    roster,  
    new CheckPerson() {  
        public boolean test(Person p) {  
            return p.getGender() == Person.Sex.MALE  
                && p.getAge() >= 18  
                && p.getAge() <= 25;  
        }  
    }  
);
```

TEMA 3: LAMBDA

- Hasta ahora hemos dado alternativas para un problema real. Como hemos visto no todas las soluciones son muy operativas y algunas generan mucho código.
- Ya hemos visto como se definen las lambdas y como funcionan los predicados. Para hacer el filtrado la más flexible posible podemos pasárselo los criterios con una lambda. Por ejemplo definiendo nuestro método así

```
public static void printPersonsWithPredicate(  
    List<Person> roster, Predicate<Person> tester) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            p.printPerson();  
        }  
    }  
}
```

- Tenemos nuestro listado de personas y un predicado sobre el cual filtraremos nuestras personas con un simple
 - `printPersonsWithPredicate(rooster, p -> p.getGender() == Person.Sex.MALE && p.getAge() >= 18 && p.getAge() <= 25)`

TEMA 3: LAMBDA

- ¿Y si quisieramos además definir la acción que queremos realizar sobre el grupo que acabamos de filtrar? Podríamos crear un método con Consumer y al lado vemos como se llama

```
public static void processPersons(  
    List<Person> roster,  
    Predicate<Person> tester,  
    Consumer<Person> block) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            block.accept(p);  
        }  
    }  
}
```

```
processPersons(  
    roster,  
    p -> p.getGender() == Person.Sex.MALE  
        && p.getAge() >= 18  
        && p.getAge() <= 25,  
    p -> p.printPerson()  
);
```

TEMA 4: STREAMS

- Los Streams nos ayudan a aplicar acciones sobre grupos de datos.
- No confundirlos con InputStream/OutputStream de Java I/O.
- Son un añadido como los lambdas de programación funcional.
- Básicamente un Stream es una secuencia de pasos, donde definimos un conjunto, operamos sobre el, filtramos y devolvemos la transformada.
- Los lambdas son una pieza clave que nos ayudará a modelar y transformar nuestros datos.

TEMA 4: STREAMS

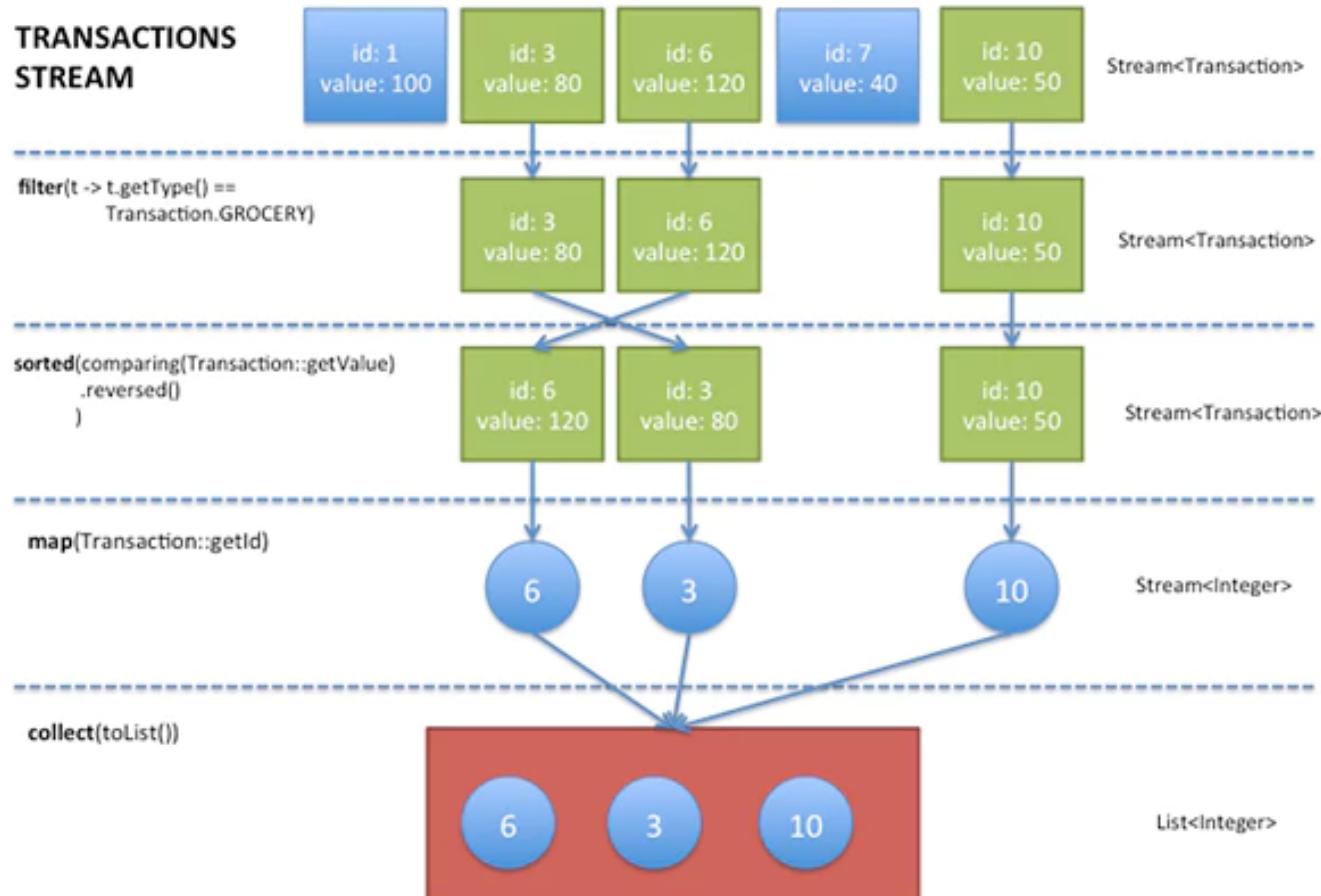
- Veamos un ejemplo

```
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList.stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

- Vemos que podemos ir encadenando acciones y obtener un resultado final.
- Vemos aquí el operador :: para invocar métodos. Si quisiéramos invocarlo directamente podemos utilizar una expresión Lambda
 - `forEach(line -> System.out.println(line))`

TEMA 4: STREAMS



TEMA 4: STREAMS

- Podemos consultar la API de los Streams
 - <https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>
- Para encadenar acciones o generar un pipeline tendremos que tener en cuenta que la acción aplicada devuelva un tipo Stream para poder seguir la cadena.
- Como hemos observado podemos usar lambdas para aplicar acciones. Debemos tener en cuenta que no deben modificar la fuente de datos y no deben tener estado (que no cambien durante la ejecución)

TEMA 4: STREAMS

- La fuente de datos puede ser generada a partir de colecciones, para ello nos da dos métodos
 - stream() que genera un stream
 - parallelStream() que veremos más adelante
- Podemos también generar nuestro Stream a través de generate o del método of

```
Stream.generate(new Random():nextDouble)  
    .limit(15)  
    .forEach(System.out::println);
```

```
Stream.of(1, 2, 3, 4)  
    .findFirst()  
    .ifPresent(System.out::println);
```

TEMA 4: STREAMS

- También podemos generar Streams de tipos específicos con
 - IntStream
 - LongStream
 - DoubleStream
- Dependiendo del tipo de Stream podemos aplicar operadores específicos, como sum() o average() para hacer operaciones

```
IntStream.range(1, 5)
          .forEach(System.out::println);
```

```
Arrays.stream(new int[] {1, 2, 3})
      .map(n -> 2 * n + 1)
      .average()
      .ifPresent(System.out::println);
```

TEMA 4: STREAMS

- Podemos ver como funciona internamente el flujo en el siguiente ejemplo

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("A");
    })
    .forEach(s -> System.out.println("forEach: " + s));

// map: d2
// filter: D2
// map: a2
// filter: A2
// forEach: A2
// map: b1
// filter: B1
// map: b3
// filter: B3
// map: c
// filter: C
```

TEMA 4: STREAMS

- Así podemos optimizar las llamadas alterando el orden

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));

// filter: d2
// filter: a2
// map: a2
// forEach: A2
// filter: b1
// filter: b3
// filter: c
```

TEMA 4: STREAMS

- Los Streams no pueden ser reutilizados, por lo que si intentamos hacerlo saltará una excepción

```
Stream<String> stream =
    Stream.of("d2", "a2", "b1", "b3", "c")
        .filter(s -> s.startsWith("a"));

stream.anyMatch(s -> true);      // ok
stream.noneMatch(s -> true);    // exception
```

- Para evitar este problema podemos usar un Stream supplier para tener todas las operaciones ya creadas

```
Supplier<Stream<String>> streamSupplier =
() -> Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> s.startsWith("a"));

streamSupplier.get().anyMatch(s -> true);    // ok
streamSupplier.get().noneMatch(s -> true);    // ok
```

TEMA 4: STREAMS

Otras funciones avanzadas son

- Collect: permite transformar un stream en un tipo diferente, por ejemplo una lista o un mapa

```
List<Persona> persons =  
    Arrays.asList(  
        new Persona("Max", "Max", 18),  
        new Persona("Peter", "Peter", 23),  
        new Persona("Pamela", "Pamela", 23),  
        new Persona("David", "David", 12));
```

```
List<Persona> filtered =  
    persons  
        .stream()  
        .filter(p -> p.getNombre().startsWith("P"))  
        .collect(Collectors.toList());  
  
System.out.println(filtered.toString());
```

```
Set<Persona> filtered =  
    persons  
        .stream()  
        .filter(p -> p.getNombre().startsWith("P"))  
        .collect(Collectors.toSet());  
  
System.out.println(filtered.toString());
```

TEMA 4: STREAMS

Otras funciones avanzadas son

- Collect: También nos permite agrupar por algún criterio

```
Map<Integer, List<Persona>> personsByAge = persons
    .stream()
    .collect(Collectors.groupingBy(p -> p.getEdad()));

personsByAge
    .forEach((age, p) -> System.out.format("age %s: %s\n", age, p));

// age 18: [Max]
// age 23: [Peter, Pamela]
// age 12: [David]
```

TEMA 4: STREAMS

Otras funciones avanzadas son

- Collect: Podemos incluso crearnos un colector para reutilizarlo más tarde

```
Collector<Persona, StringJoiner, String> personNameCollector =  
    Collector.of(  
        () -> new StringJoiner(" | "), // supplier  
        (j, p) -> j.add(p.getNombre().toUpperCase()), // accumulator  
        (j1, j2) -> j1.merge(j2), // combiner  
        StringJoiner::toString); // finisher  
  
String names = persons  
    .stream()  
    .collect(personNameCollector);  
  
System.out.println(names); // MAX | PETER | PAMELA | DAVID
```

TEMA 4: STREAMS

Otras funciones avanzadas son

- FlatMap: nos permite transformar todos los elementos de un stream en un stream de otro tipo de objetos. Por ejemplo para convertir un Stream de un tipo a otro stream de un tipo que extienda (y por consiguiente necesita más campos)
- Reduce: Nos permite ir reduciendo un Stream hasta que concuerde con nuestro criterio aceptando un operador binario

```
persons
    .stream()
    .reduce((p1, p2) -> p1.getEdad() > p2.getEdad() ? p1 : p2)
    .ifPresent(System.out::println);      // Pamela
```

TEMA 4: STREAMS

Parallel Streams

- Una de las ventajas de los Streams es que nos permiten procesar conjuntos de una manera sencilla y rápida.
- Es posible que tengamos que procesar cargas grandes de streams, para ello podemos hacer streams paralelos.

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
.parallelStream()
.filter(s -> {
    System.out.format("filter: %s [%s]\n",
        s, Thread.currentThread().getName());
    return true;
})
.map(s -> {
    System.out.format("map: %s [%s]\n",
        s, Thread.currentThread().getName());
    return s.toUpperCase();
})
.forEach(s -> System.out.format("forEach: %s [%s]\n",
    s, Thread.currentThread().getName()));
```

```
filter: a1 [ForkJoinPool.commonPool-worker-2]
filter: b1 [main]
filter: a2 [ForkJoinPool.commonPool-worker-1]
map: a2 [ForkJoinPool.commonPool-worker-1]
forEach: A2 [ForkJoinPool.commonPool-worker-1]
filter: c2 [ForkJoinPool.commonPool-worker-1]
map: c2 [ForkJoinPool.commonPool-worker-1]
forEach: C2 [ForkJoinPool.commonPool-worker-1]
filter: c1 [ForkJoinPool.commonPool-worker-3]
map: c1 [ForkJoinPool.commonPool-worker-3]
forEach: C1 [ForkJoinPool.commonPool-worker-3]
map: b1 [main]
map: a1 [ForkJoinPool.commonPool-worker-2]
forEach: A1 [ForkJoinPool.commonPool-worker-2]
forEach: B1 [main]
```

TEMA 4: STREAMS

Parallel Streams

- Por norma general usar procesamiento paralelo mejorará el rendimiento
- Pero hay excepciones, ya que hay operaciones que necesitan procesamiento adicional que combine hilos de ejecución como sort() por lo que debemos tener cuidado.
- Por norma general para grandes colecciones puede ser muy útil.

TEMA 5: MEJORAS EN APIs

- Es muy importante estar familiarizado con el manejo de cadenas String.
- Para la concatenación es muy importante el orden siempre aplicado de izquierda a derecha.
- Es importante tener claro la inmutabilidad de las cadenas. Una vez que generamos una cadena esta no cambia gracias al pool de strings, aunque solo las que se generan con de manera *String cadena = “una cadena”;*

TEMA 5: MEJORAS EN APIs

- Métodos importantes de String:
 - length()
 - charAt()
 - indexOf()
 - substring(), también con parámetros. int substring(int beginIndex, int endIndex) el end index no se incluye y puede llegar a la longitud total de la cadena. Es importante vigilar los StringIndexOutOfBoundsException
 - toLowerCase() y toUpperCase()
 - equals() y equalsIgnoreCase()
 - startsWith() y endsWith()
 - contains()
 - String replace(char oldChar, char newChar) y String replace(CharSequence oldChar, CharSequence newChar)
 - trim(), no solo quita los espacios del inicio y final de la cadena, también elimina los \t (tabulador), \n (nueva línea) y \r (retorno de carro)



TEMA 5: MEJORAS EN APIs

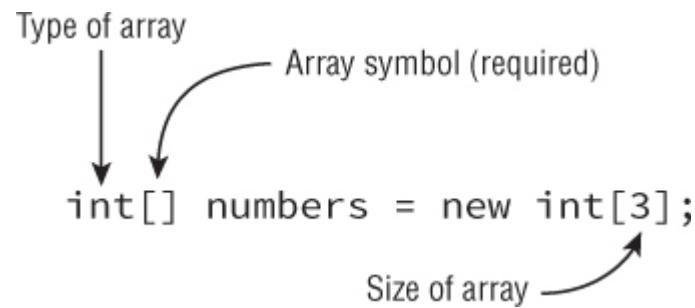
- `StringBuilder` nos permite crear cadenas mutables fácilmente. Debemos conocer los siguientes métodos
 - `charAt()`, `indexOf()`, `length()`, y `substring()` funcionan igual que los de `String`
 - `append()`
 - `insert(int offset, String str)` offset es a partir de cuando queremos insertar

```
3: StringBuilder sb = new StringBuilder("animals");
4: sb.insert(7, "-");                                // sb = animals-
5: sb.insert(0, "-");                                // sb = -animals-
6: sb.insert(4, "-");                                // sb = -ani-mals
7: System.out.println(sb);
```

- `StringBuilder delete(int start, int end)` y `StringBuilder deleteCharAt(int index)` el end es no se incluye en el borrado.
- `reverse()`
- `toString()`

TEMA 5: MEJORAS EN APIs

- StringBuffer hace lo mismo pero es un poco más lento porque es seguro en hilos de ejecución, aunque esto no es necesario saberlo. Normalmente sólo habrá preguntas sobre StringBuilder.
- Es importante tener muy claro como funcionan los operadores de igualdad. Sobre todo para las cadenas que se generan en el pool de strings. Es importante recordar que debe usarse .equals() en nuestro código.
- Creación de arrays



TEMA 5: MEJORAS EN APIs

- También podemos crear arrays especificando los elementos
 - `int[] numbers2 = new int[] {42, 55, 99};`
 - `int[] numbers2 = {42, 55, 99};`
- Java deja indicar que es un array delante o detrás del nombre de la variable.
- Para la declaración múltiple `int ids[], types;` crea un array y un int.
- Al hacer un `toString()` de un array tenemos resultados de este tipo
`[Ljava.lang.String;@160bc7c0`
- Podemos imprimir con `java.util.Arrays.toString(miArray)`
- Los arrays atributos de una clase se inicializan a null, es importante vigilar su inicialización.
- También se pueden utilizar en castings (`String[]`) aunque debemos tener cuidado de no almacenar objetos que no concuerden con el array.
- Hay que vigilar los `ArrayIndexOutOfBoundsException` siempre que veamos arrays



TEMA 5: MEJORAS EN APIs

- Arrays.sort(miArray); permite ordenar un array.
- Arrays.binarySearch(miArray, elementoABuscar) permite hacer una búsqueda en un array. Para ello debe estar previamente ordenado. El array que no esté ordenado dará un resultado imprevisible. Si el elemento no está en el array indica en qué posición debería insertarse.
- Los varargs son tratados como arrays public static void main(String... args)
// varargs
- Hay que dominar los arrays multidimensionales
 - int[][] vars1; // 2D array
 - int vars2 [][]; // 2D array
 - int[] vars3[]; // 2D array
 - int[] vars4 [], space [][]; // a 2D AND a 3D array
 - String [][] rectangle = new String[3][2];
 - int[][] differentSize = {{1, 4}, {3}, {9,8,7}};



TEMA 5: MEJORAS EN APIs

- ArrayList también es muy popular.
 - ArrayList list1 = new ArrayList();
 - ArrayList<String> list4 = new ArrayList<String>();
 - Es importante recordar que ArrayList implementa List, pero que no podemos instanciar List directamente.
 - add()
 - remove()
 - set()
 - isEmpty() y size()
 - clear()
 - contains()
 - equals()



TEMA 5: MEJORAS EN APIs

- Wrappers

Primitive type	Wrapper class	Example of constructing
boolean	Boolean	<code>new Boolean(true)</code>
byte	Byte	<code>new Byte((byte) 1)</code>
short	Short	<code>new Short((short) 1)</code>
int	Integer	<code>new Integer(1)</code>
long	Long	<code>new Long(1)</code>
float	Float	<code>new Float(1.0)</code>
double	Double	<code>new Double(1.0)</code>
char	Character	<code>new Character('c')</code>

TEMA 5: MEJORAS EN APIs

- Conversiones a string

Wrapper class	Converting string to primitive	Converting string to wrapper class
Boolean	Boolean.parseBoolean("true");	Boolean.valueOf("TRUE");
Byte	Byte.parseByte("1");	Byte.valueOf("2");
Short	Short.parseShort("1");	Short.valueOf("2");
Integer	Integer.parseInt("1");	Integer.valueOf("2");
Long	Long.parseLong("1");	Long.valueOf("2");
Float	Float.parseFloat("1");	Float.valueOf("2.2");
Double	Double.parseDouble("1");	Double.valueOf("2.2");
Character	None	None



TEMA 5: MEJORAS EN APIs

- Java permite convertir cualquier valor primitivo en su wrapper automáticamente. Es un proceso llamado autoboxing. Debemos vigilar con los NullPointerException ya que es válido la asignación de nulos a un wrapper pero al intentar hacer el autoboxing fallará en tiempo de ejecución.
- Conversiones entre listas y arrays
 - `Object[] objectArray = list.toArray();`
 - También podemos especificar el tipo al que convierte `String[] stringArray = list.toArray(new String[0]);`
 - También hay un método a la inversa en `java.util.Arrays`. `List<String> list = Arrays.asList(array);`
- Ordenación de `ArrayList` a través de un método `Collections.sort(miLista);`

TEMA 5: MEJORAS EN APIs

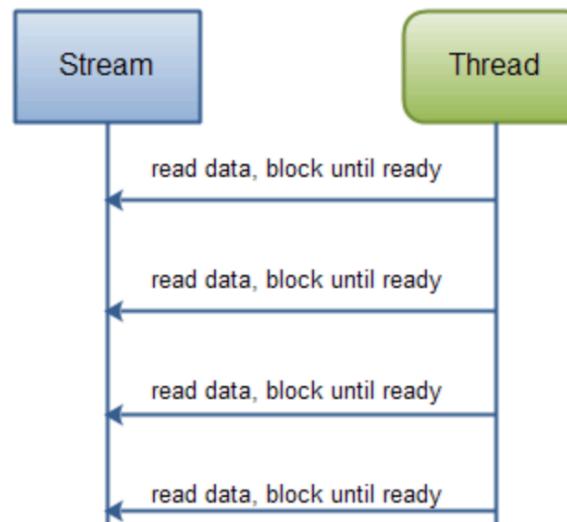
Java NIO.2

- En Java 7 se introdujeron mejoras en la gestión de ficheros. Para ello se redefinió la librería NIO (o non-blocking I/O) en NIO.2
- java.io sigue siendo perfectamente válida, estas son las diferencias
 - Java.io esta orientada a streams mientras que NIO.2 está orientada a buffers. Los streams (no confundir con los Streams del tema 4) una vez consumidos no pueden volverse a tratar mientras que los buffers si.
 - Java.io bloquea los streams en tanto que NIO.2 no bloquea los recursos.
 - NIO.2 permite gestionar en un solo hilo de ejecución varios canales de entrada.
- Para saber que implementación utilizar normalmente usaremos Java IO si necesitamos pocas conexiones con un buen ancho de banda.
- Usaremos Java NIO.2 cuando necesitemos gestionar varios canales a la vez en un hilo. También si hay poca transferencia de datos pero muchas conexiones es más ventajoso (una aplicación P2P)

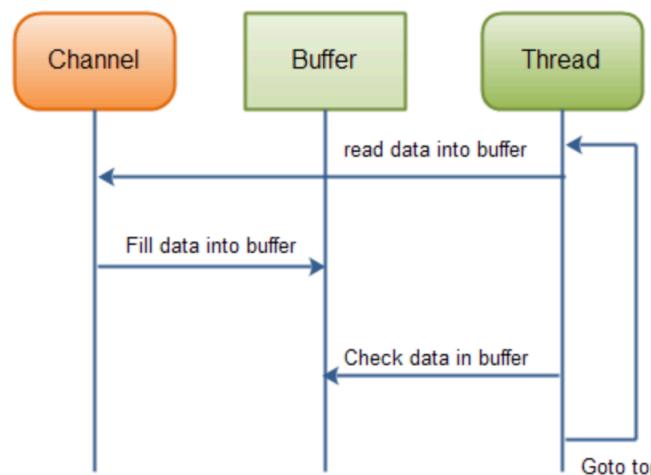
TEMA 5: MEJORAS EN APIs

Java NIO.2

- Ejecución de Java IO



- Ejecución de Java NIO.2



TEMA 5: MEJORAS EN APIs

Java NIO.2

- Para saber que implementación utilizar normalmente usaremos Java IO si necesitamos pocas conexiones con un buen ancho de banda.
- Usaremos Java NIO.2 cuando necesitemos gestionar varios canales a la vez en un hilo. También si hay poca transferencia de datos pero muchas conexiones es más ventajoso (una aplicación P2P)

TEMA 5: MEJORAS EN APIs

Java NIO.2

- Para gestionar rutas tenemos la clase Path

```
private static String HOME = System.getProperty("user.home");

public static void main (String[] args){
    Path p = Paths.get(HOME);

    System.out.println(Files.exists(p));
}
```

- Con Files podemos gestionar los ficheros
 - exists() para comprobar la existencia de un fichero
 - notExists()
 - isRegularFile() para comprobar si es fichero o directorio
 - isReadable() para comprobar si podemos leer
 - isWritable()
 - isExecutable()

TEMA 5: MEJORAS EN APIs

Java NIO.2

- Files
 - `createFile()` para crear un fichero
 - `createDirectory()` para crear un directorio
 - `createTempFile()` para crear un fichero temporal
 - `delete()` para borrar un fichero o directorio (si está vacío)
 - `copy()` para copiar ficheros
 - `move()` para trasladar ficheros
 - `readAllLines()` para leer un fichero
 - `lines()` crea un stream con las líneas de nuestro fichero
 - `write()` nos permite escribir bytes en un fichero

TEMA 5: MEJORAS EN APIs

Java NIO.2 Escritura de ficheros

- A pesar de que Files.write() nos permite escribir en ficheros no es la manera óptima para escribir ficheros grandes. Para ello podemos usar BufferedWriter

```
try (BufferedWriter writer = Files.newBufferedWriter(path, Charset.forName("UTF-8"))) {  
    writer.write("Escribiendo con BufferedWriter\n");  
} catch (IOException ex) {  
    ex.printStackTrace();  
}
```

TEMA 5B: TIPOS GENÉRICOS

- En ocasiones podemos crear clases que gestionen otras clases. Por ejemplo, una Lista, aunque con entidad propia no entra a realizar acciones sobre los objetos que alberga.
- Java nos da un mecanismo para poder gestionar este tipo de clases. Los tipos genéricos.
- Básicamente declaramos que una clase puede albergar cualquier tipo de objeto.
- Veamos un ejemplo

TEMA 5B: TIPOS GENÉRICOS

- Supongamos que queremos hacer una clase caja para mapear objetos de cualquier instancia en nuestro código

```
public class Caja {  
  
    private Object objeto;  
  
    public Object get() {  
        return objeto;  
    }  
  
    public void set(Object objeto) {  
        this.objeto = objeto;  
    }  
}
```

TEMA 5B: TIPOS GENÉRICOS

- Aunque podríamos meter cualquier cosa dentro de esa clase tenemos el problema de tener que estar operando con castings todo el tiempo para asegurarnos de que el compilador deja trabajar, además no podemos controlar si se meten instancias de distintas clases.
- Para ello podemos convertirla en genérica

```
public class Caja<T> {  
  
    private T t;  
  
    public T get() {  
        return t;  
    }  
  
    public void set(T t) {  
        this.t = t;  
    }  
}
```

TEMA 5B: TIPOS GENÉRICOS

- Para utilizarla tan solo tendríamos que hacer la llamada

```
Caja<String> c = new Caja<>();
```

- Si queremos utilizar más tipos diferentes en nuestra clase tan solo tenemos que indicarlo en la notación class

```
public class Caja<K, V>{}
```

- Por convención los nombre usados son los siguientes

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

TEMA 6: TIME

- Java 8 incluye un nuevo manejo de fechas más natural. Para ello debemos conocer el paquete `java.time`.^{*}
 - Hay tres tipos de creación de fechas
 - `LocalDate`: solo una fecha, sin hora y sin timezone
 - `LocalTime`: solo una hora, sin fecha y sin timezone
 - `LocalDateTime`: fecha y hora sin timezone.
 - Debemos conocer que el formato de fecha en EEUU es `mm/dd/yyyy`
 - Las fechas son inmutables
 - Veamos una comparativa

TEMA 6: TIME

	Old way	New way (Java 8 and later)
Importing	<code>import java.util.*;</code>	<code>import java.time.*;</code>
Creating an object with the current date	<code>Date d = new Date();</code>	<code>LocalDate d = LocalDate.now();</code>
Creating an object with the current date and time	<code>Date d = new Date();</code>	<code>LocalDateTime dt = LocalDateTime.now();</code>
Creating an object representing January 1, 2015	<pre>Calendar c = Calendar.getInstance(); c.set(2015, Calendar.JANUARY, 1); Date jan = c.getTime(); or Calendar c = new GregorianCalendar(2015, Calendar.JANUARY, 1); Date jan = c.getTime();</pre>	<code>LocalDate jan = LocalDate.of(2015, Month.JANUARY, 1);</code>
Creating January 1, 2015 without the constant	<pre>Calendar c = Calendar.getInstance(); c.set(2015, 0, 1); Date jan = c.getTime();</pre>	<code>LocalDate jan = LocalDate.of(2015, 1, 1)</code>



TEMA 6: TIME

- Veamos que métodos podemos llamar en cada clase

	Can call on <code>LocalDate</code> ?	Can call on <code>LocalTime</code> ?	Can call on <code>LocalDateTime</code> ?
<code>plusYears/minusYears</code>	Yes	No	Yes
<code>plusMonths/minusMonths</code>	Yes	No	Yes
<code>plusWeeks/minusWeeks</code>	Yes	No	Yes
<code>plusDays/minusDays</code>	Yes	No	Yes
<code>plusHours/minusHours</code>	No	Yes	Yes
<code>plusMinutes/minusMinutes</code>	No	Yes	Yes
<code>plusSeconds/minusSeconds</code>	No	Yes	Yes
<code>plusNanos/minusNanos</code>	No	Yes	Yes

TEMA 6: TIME

- Y por último como manipular fechas

	Old way	New way (Java 8 and later)
Adding a day	<pre>public Date addDay(Date date) { Calendar cal = Calendar.getInstance(); cal.setTime(date); cal.add(Calendar.DATE, 1); return cal.getTime(); }</pre>	<pre>public LocalDate addDay(LocalDate date) { return date.plusDays(1); }</pre>
Subtracting a day	<pre>public Date subtractDay(Date date) { Calendar cal = Calendar.getInstance(); cal.setTime(date); cal.add(Calendar.DATE, -1); return cal.getTime(); }</pre>	<pre>public LocalDate subtractDay(LocalDate date) { return date.minusDays(1); }</pre>

TEMA 6: TIME

- También podemos trabajar con períodos de tiempo con la clase Period. Hay que vigilar las concatenaciones

```
public static void main(String[] args) {
    LocalDate start = LocalDate.of(2015, Month.JANUARY, 1);
    LocalDate end = LocalDate.of(2015, Month.MARCH, 30);
    Period period = Period.ofMonths(1);           // create a period
    performAnimalEnrichment(start, end, period);
}
private static void performAnimalEnrichment(LocalDate start, LocalDate end,
    Period period) {                         // uses the generic period
    LocalDate upTo = start;
    while (upTo.isBefore(end)) {
        System.out.println("give new toy: " + upTo);
        upTo = upTo.plus(period);      // adds the period
    }
}
```

- Se pueden crear de diversas maneras

```
Period annually = Period.ofYears(1);          // every 1 year
Period quarterly = Period.ofMonths(3);         // every 3 months
Period everyThreeWeeks = Period.ofWeeks(3);     // every 3 weeks
Period everyOtherDay = Period.ofDays(2);        // every 2 days
Period everyYearAndAWeek = Period.of(1, 0, 7);   // every year and 7 days
```

TEMA 6: TIME

- Period es para períodos a partir de un día. Para menores escalas de tiempo tenemos Duration que es para días o menos.
- Period no puede usarse con LocalTime (recordad que solo indicaba una hora) y si encadenamos periodos solo se tiene en cuenta el último.
- Para formatear fechas disponemos de DateTimeFormatter de los que solo necesitamos conocer los SHORT y MEDIUM

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dateTime = LocalDateTime.of(date, time);
DateTimeFormatter shortF = DateTimeFormatter
    .ofLocalizedDateTime(FormatStyle.SHORT);
DateTimeFormatter mediumF = DateTimeFormatter
    .ofLocalizedDateTime(FormatStyle.MEDIUM);
System.out.println(shortF.format(dateTime));      // 1/20/20 11:12 AM
System.out.println(mediumF.format(dateTime));      // Jan 20, 2020 11:12:34 AM
```

TEMA 6: TIME

- También podemos generar uno personalizado DateTimeFormatter f = DateTimeFormatter.ofPattern("MMMM dd, yyyy, hh:mm");
- Para parsear fechas debemos conocer el método parse

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("MM dd yyyy");
LocalDate date = LocalDate.parse("01 02 2015", f);
LocalTime time = LocalTime.parse("11:22");
System.out.println(date);           // 2015-01-02
System.out.println(time);          // 11:22
```