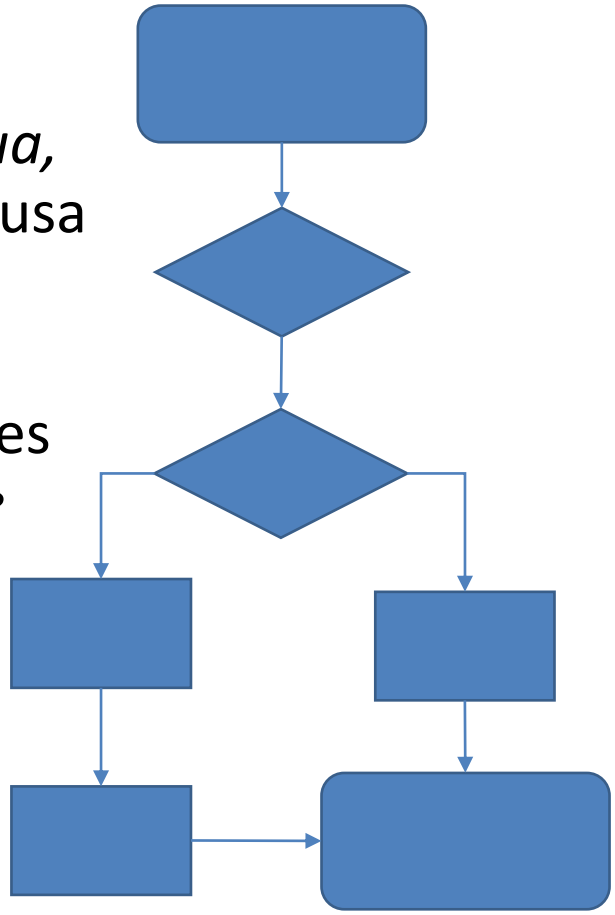
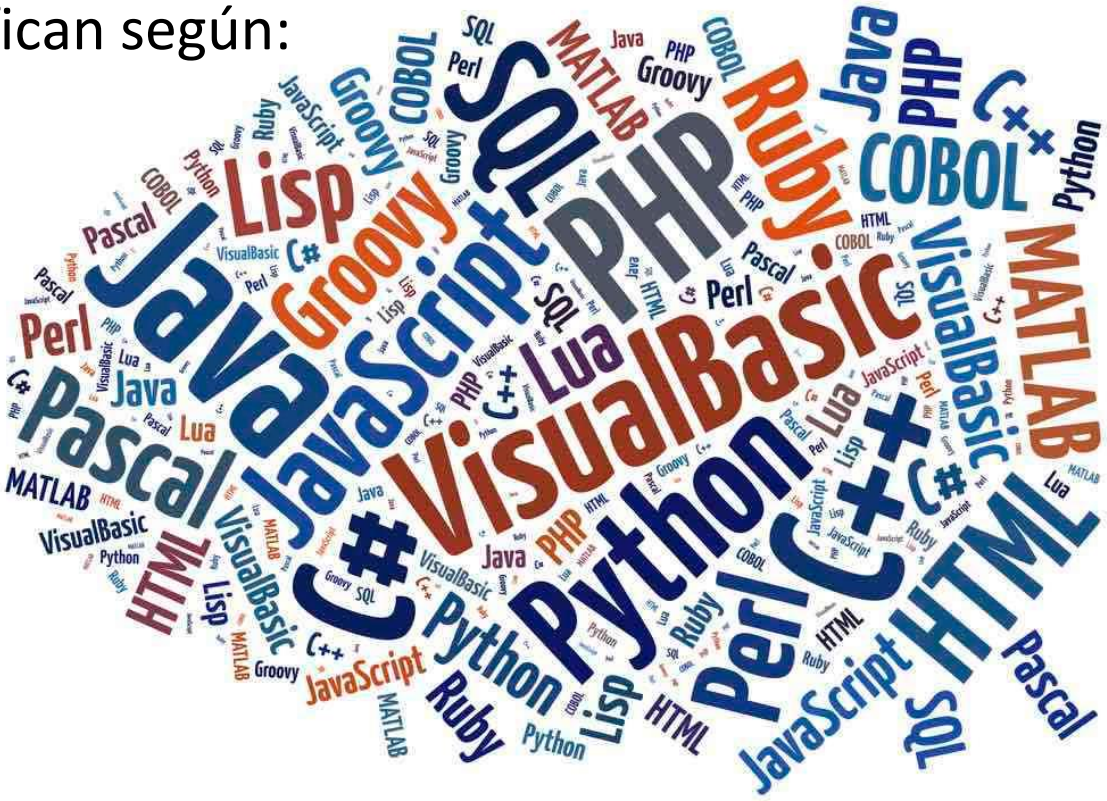

imatia
innovation
We help you to do more

- Introducción a la programación.
- Tipos, variables, constantes, asignaciones.
- Funciones, procedimientos y métodos
- Definición de clase, objeto e interfaz.
- Estructuras de control.
- Estructuras de almacenamiento.
- Lectura y escritura de información.

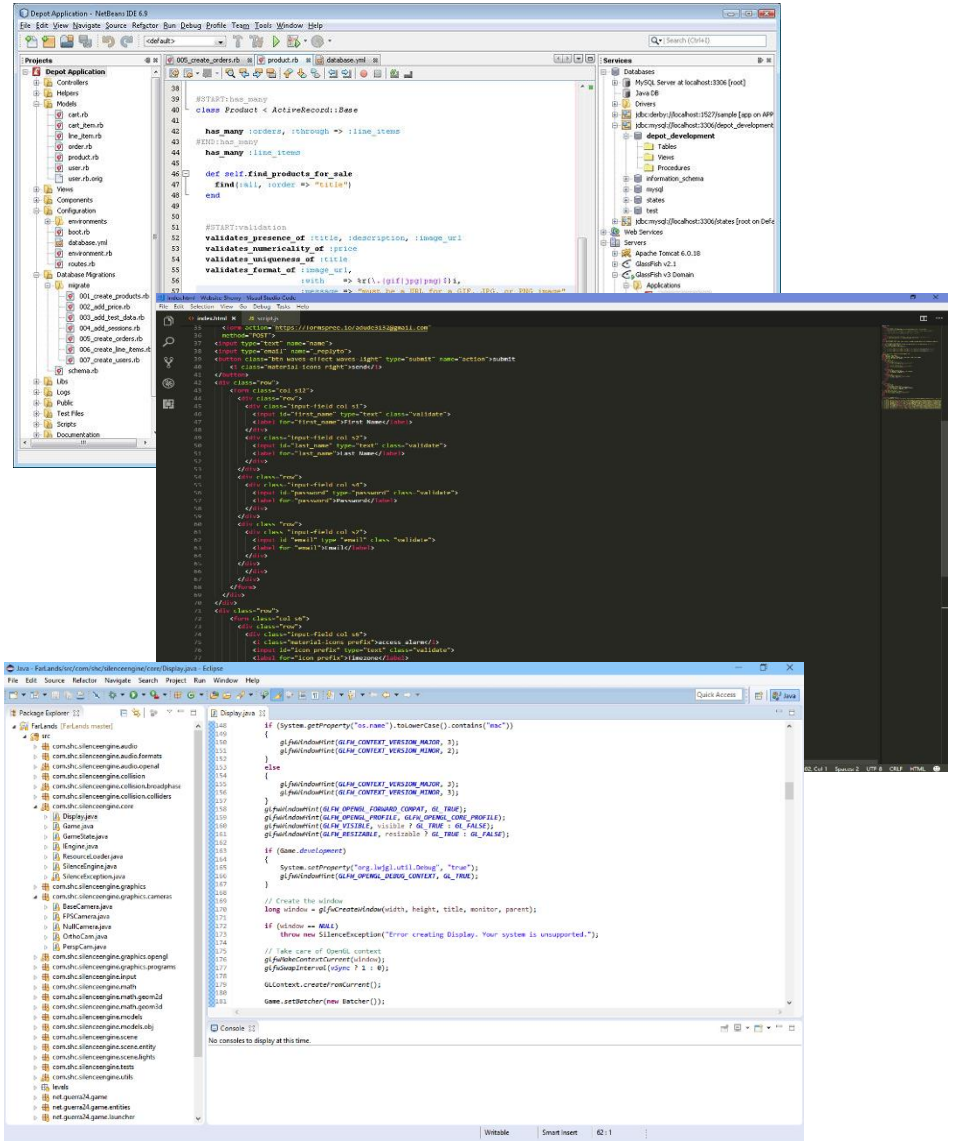
- La **programación** se define como la *implementación de un algoritmo* para realizar un **programa.**
- Un **algoritmo** es una secuencia *no ambigua, finita y ordenada* de instrucciones que se usa para resolver un problema.
- La **programación** tiene como objetivo la creación de **programas**, cuyas instrucciones un ordenador *es capaz de interpretar y de ejecutar.*



- Para crear un programa, empleamos lenguajes de programación.
- Existen múltiples lenguajes de programación en la actualidad.
- Los lenguajes se clasifican según:
  - Nivel:
    - De alto nivel.
    - De bajo nivel.
  - Tipo ejecución.
    - Compilados.
    - Interpretados.



- Para programar, es común usar un entorno de *desarrollo integrado,* comúnmente denominado **IDE.**
- Os **IDE** son programas que facilitan el proceso de desarrollo de aplicaciones y incluyen interfaces gráficas.
- Existen diferentes **IDE**s, como por ejemplo, Eclipse, NetBeans, Visual Studio Code…



- Las variables son los datos con los que trabajan nuestros programas.
- Estas variables almacenan la información en la memoria, e permiten o acceso a dichos datos.
- As variables se declaran dando un nombre a un tipo de dato

horas = 2.5

mes = 12

nombre = “Imatia”

- Los lenguajes de programación poden ser tipados o no tipados.
- Java es un lenguaje tipado.
- Hay que indicar el tipo de dato al que pertenece la variable.
- Ls tipos poden ser primitivos o de tipo objeto.

```
long horas = 2.5
```

```
int mes = 12
```

```
String nombre = “Imatia”
```

- Diferente a las variables, las constantes permiten almacenar un valor invariable y fijo durante la ejecución de un programa.
- La declaración de constante, a mayores del tipo de dato, es necesaria por ser un lenguaje tipado, incluye la palabra reservada *final.*

Variable	Constante
int primeiroDia = 1	final int PRIMEIRO_DIA = 1

Tipos de datos

Tipos primitivos	Tipos objeto			
byte, short, int, long, float, double, char, boolean	Tipos de la biblioteca estándar de Java	Tipos definidos por el programador	arrays	Tipos envoltorio
	String, Scanner, TreeSet, ArrayList…	***Clases propias: Animal, Trabajador, Lámpara…***	***Series de elementos, como vectores ou matrices***	***Byte, Short, Integer, Long, Float, Double, Character, Boolean***

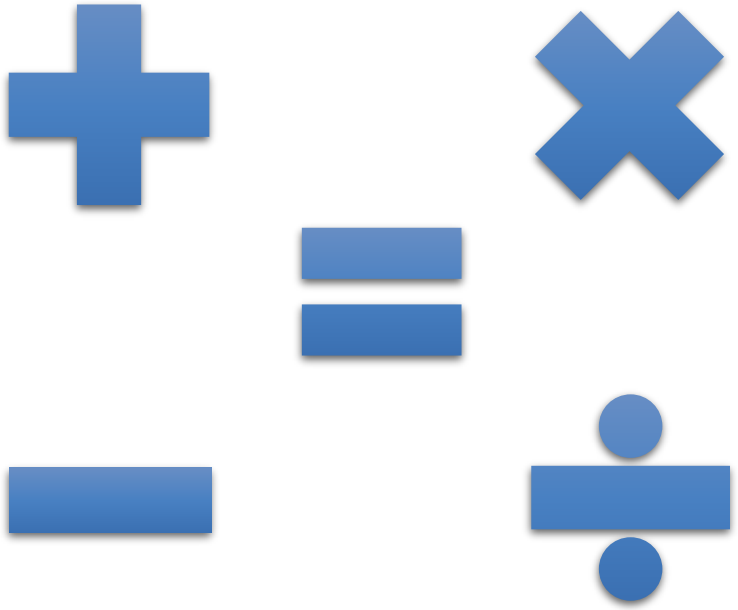
- Se pueden asignar valores a las variables y trabajar con ellos, utilizando o símbolo de igual (=).
- Cambia o valor da variable da izquierda por el resultado da expresión de la derecha

```
int valor = 5;                  // 5
valor = valor + 3;              // 8
valor = 2 - valor;              // -6
valor += 3;                     // -3
valor -= 1;                     // -4
valor *= 4;                     // -16
valor = (valor / 2) + 1;        // -7
int nuevo = 3;                  // 3
int miValor = 2;                // 2
valor = nuevo + miValor;        // 5
final int CONST = 2;            // 2
valor = CONST;                  // 2
valor += 3;                     // 5
```

- Algunos elementos pueden ter múltiples representaciones, como cadena, entero o doble.
- Para pasar de un tipo a otro hay que hacer una conversión (cast).

```
String cnv = "200";                                 // class java.lang.String
Integer cnv_int = Integer.parseInt(cnv);            // class java.lang.Integer
String cnv_str = Integer.toString(cnv_int);         // class java.lang.String
Character cnv_chr = cnv.charAt(0);                  // class java.lang.Character
String cnv_chr_str = Character.toString(cnv_chr);   // class java.lang.String
Double cnv_dbl = Double.parseDouble(cnv_str);       // class java.lang.Double
String cnv_dbl_str = String.valueOf(cnv_dbl);       // class java.lang.String
Float cnv_flt = Float.parseFloat(cnv);              // class java.lang.Float
String cnv_flt_str = Float.toString(cnv_flt);       // class java.lang.String
Boolean cnv_bool = Boolean.valueOf("true");         // class java.lang.Boolean
String cnv_bool_str = String.valueOf(cnv_bool);     // class java.lang.String
```

- Un operador es un símbolo que se emplea para modificar el valor de las variables afectadas
- Existen 3 tipos de operadores
  - Operadores aritméticos
  - Operadores lógicos
  - Operadores a nivel de bit



- Los operadores más comunes son los aritméticos y los operadores lógicos

- Operadores aritméticos

Operador	Operación	Ejemplo
+	Suma	4 + 2
-	Resta	4 – 2
*	Multiplicación	4 * 2
/	División	4 / 2
%	Módulo	4 % 2
++	Incremento	4++
--	Decremento	4++

- Operadores lóxicos

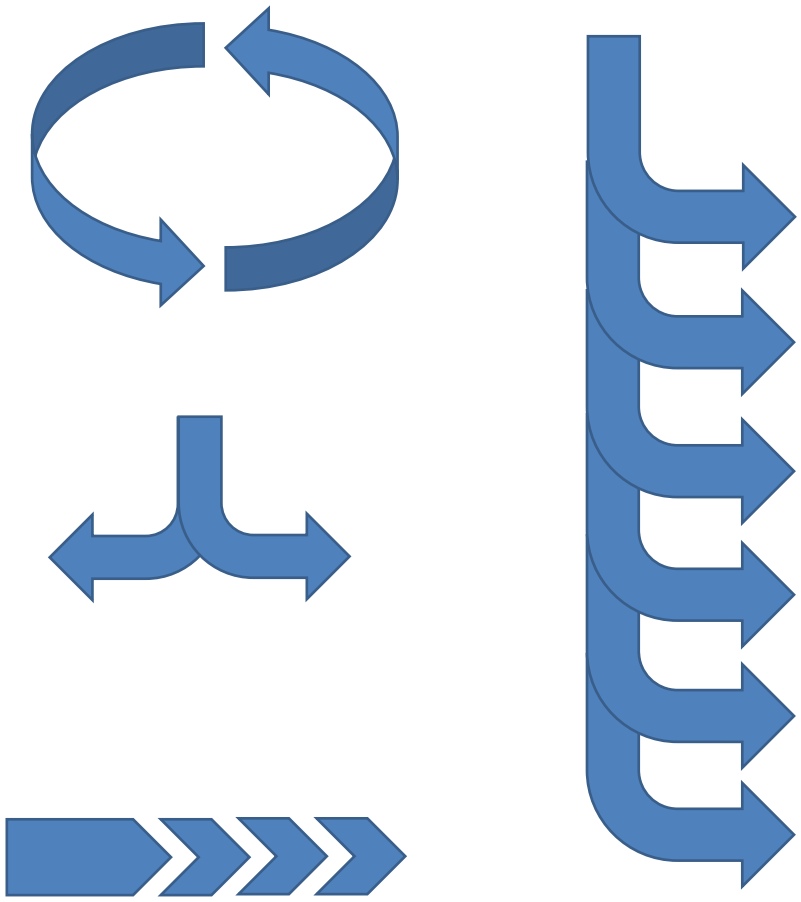
Operador	Operación	jxemplo
==	Igual	4 == 4
!=	Distinto	3 != 4
<, <=, >, >=	Menor, menor o igual, mayor, mayor o igual	3 > 5 5 <= 5
&&	Operador AND (y)	1 && 0
||	Operador OR (o)	0 || 1
!	Operador NOT (no)	!1

- Una función es una serie de instrucciones, encapsuladas, que pueden recibir una serie de valores para ejecutar operaciones y devolver un valor.

- Un procedimiento es una serie de instrucciones, encapsuladas, que pueden recibir una serie de valores para ejecutar operaciones que non devolverán ningún valor.

- **Un método es una serie de instrucciones, encapsuladas, que poden recibir una serie de valores para ejecutar operaciones y puede devolver un valor.**

- Las estructuras de control sirven para modificar el flujo de ejecución de un programa
- Existen estructuras de selección y estructuras iterativas, que pueden anidarse entre ellas



- Las estructuras de control de selección permiten modificar el flujo de un programa según un determinado esquema
- La estructura ***if*** sirve para comprobar una condición, y en base a ella, ejecutar o no una porción de código

```
SI (Condición a cumprir) {
   //Código se a condición se cumpre
}
```

```
SI (Condición a cumprir) {
   //Código se a condición se cumpre
}SI NO {
   //Código se a condición non se cumpre
}
```

```
SI (Condición 1 a cumprir) {
   //Código se a condición 1 se cumpre
   } SI NO SI (Condición 2 a cumprir){
      //Código se a condición 2 se cumpre
   } SI NO {
      //Código se a condición 1 e 2 non se cumpren
}
```

```
int value = 4;

if (value >= 5) {
    System.out.println("Maior ou igual a 5");
}else {
    System.out.println("Menor que 4");
}
```

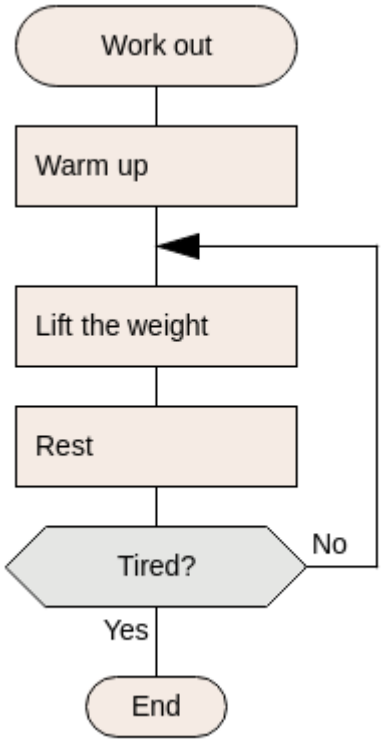
- Otra estructura, llamada ***switch***, sirve para indicar más de dos alternativas

```
switch (expresión){
case valor_expr:
    …
    break;
case valor_expr2:
    …
    break;
case valor_expr3:
    …
    break;
default:
    …
    break;
}
```

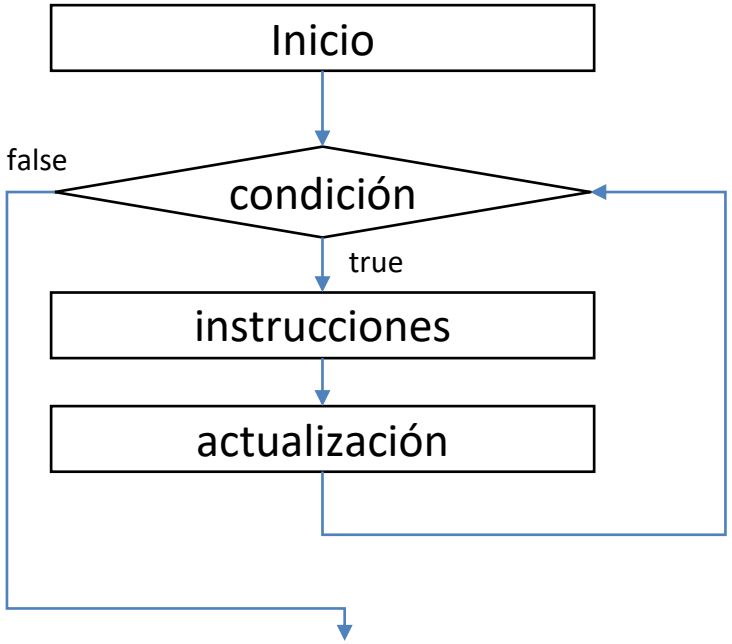
```
int value = 2;

switch (value + 1) {
case 1:
    value = value + 1;
    break;
case 2:
    value = value + 0;
    break;
case 3:
    value = value - 1;
    break;
default:
    value = value * 2;
    break;
}
```

- Se denominan estructuras iterativas o **bucles** aquellas estructuras que permiten ejecutar múltiples veces una sección de código

- Un ejemplo de un bucle representado gráficamente



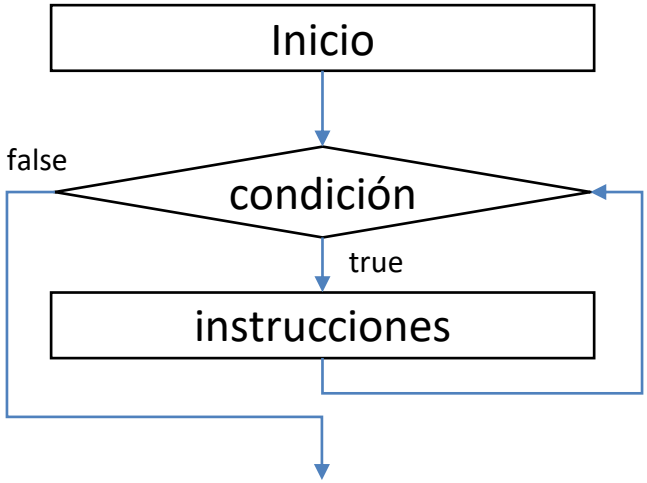
- El bucle **for** se utiliza para realizar la ejecución de un código un número determinado de veces




```
for (inicialización;condición;actualización){
    instrucciones
}
```

```
for (int i = 0; i < 10; i++) {
    System.out.println("O número é: " + (i + 1));
}
```

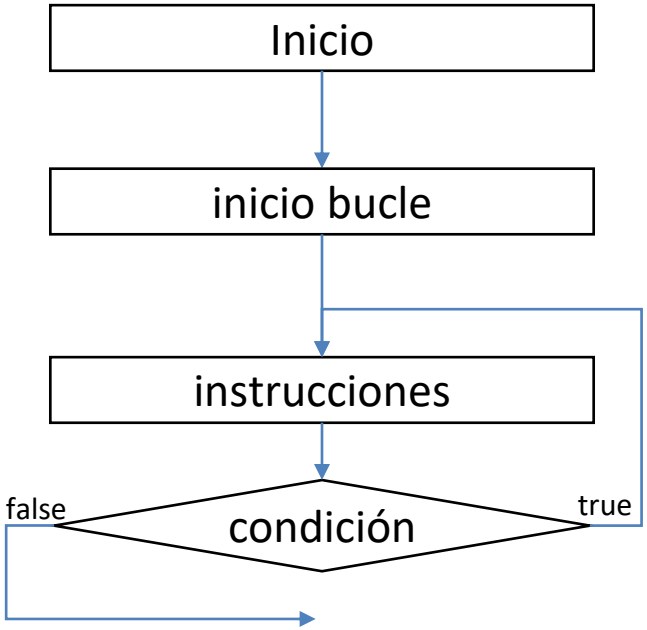
- El bucle **while** se utiliza para realizar la ejecución de un código mientras la condición sea verdadera




```
while (condición){
    instrucciones
}
```

```
int i = 0;

while (i < 11) {
    i++;
}
```

- El bucle **do-while** se utiliza para realizar la ejecución de un código mientras la condición sea verdadera, pero se ejecuta como mínimo una vez




```
do {
    instruccións
} while (condición)
```

- Los bucles poden ocasionar que se cree un **bucle infinito**

- La sentencia **break** permite romper a ejecución de un bucle (o un switch), y o control pase a primera instrucción fuera de la estructura del control de flujo

- La sentencia **continue** salta automáticamente a la siguiente iteración de la estructura de control.

- **break**

```
int i = 0;

while (i <= 10) {
    i++;
    if ((i % 5) == 0) {
        break;
    }
    System.out.println("El valor es: "+ i);
}
System.out.println("Fin del bucle.");
```

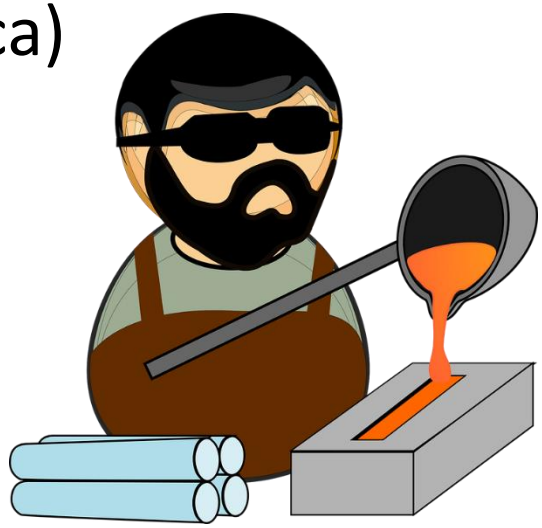
- Antes de que se imprima el mensaje de que el valor es 5, el bucle acabará.

- **continue**

```
int i = 0;

while (i <= 10) {
    i++;
    if ((i % 2) == 0) {
        System.out.println("El valor es:" + i);
    }
}
```

- Solo se imprime el mensaje cada vez que el valor de **i** sea par.

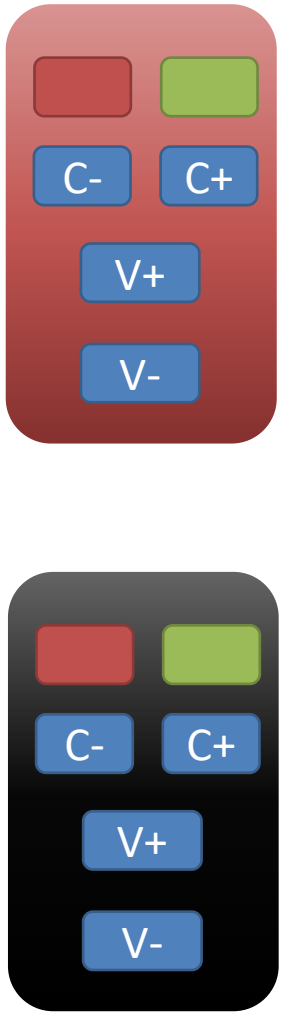
- Las clases y los objetos son los pilares principales de la programación orientada a objetos
- Un símil para entender las clases y los objetos: Las clases son la representación de los elementos del mundo real (clase Persona), y los objetos son elementos únicos que pertenecen a esa representación (una persona específica)
- Todos los objectos de una clase tienen las mismas cualidades pero con diferente valor (atributos), y pueden hacer as mismas acciones (métodos)



```java
public class TVRemote {

    int channel;
    int volume;
    boolean on;
    String color;

    public TVRemote(String color) {
        this.channel = 0;
        this.volume = 20;
        this.color = color;
    }

    public boolean turnOn() {
        this.on = true;
        return this.on;
    }

    public boolean turnOff() {
        this.on = false;
        return this.on;
    }

    public void channelUp() {
        this.channel++;
    }

    public void channelDown() {
        if (this.checkMinValue(this.channel)) {
            this.channel--;
        }
    }

    public void volumeUp() {
        this.volume++;
    }

    public void volumeDown() {
        if (this.checkMinValue(this.volume)) {
            this.volume--;
        }
    }

    public String getColor() {
        return this.color;
    }

    private boolean checkMinValue(int value) {
        if (value == 0) {
            return false;
        } else {
            return true;
        }
    }
}
```

- A la izquierda, un ejemplo de una clase de mando a distancia muy simple, donde podemos apreciar los atributos y los métodos
- A la derecha y debajo, la creación de los objetos utilizando a clase.

```java
public static void main(String[] args) {
    TVRemote redRemote = new TVRemote("Vermello");
    TVRemote blackRemote = new TVRemote("Negro");
    System.out.println(redRemote.getColor());
    System.out.println(blackRemote.getColor());
}
```



- El método con el mismo nombre que la clase se llama “constructor” y sirve para crear un nuevo objeto, una nueva instancia de la clase.

```
public TVRemote(String color) {
    this.channel = 0;
    this.volume = 20;
    this.color = color;
}
```

- Este es el método que debemos llamar para crear el objeto.
- Usando la palabra reservada *new* y el constructor, creamos una nueva instancia.
- O constructor puede tener cualquier número de parámetros

- Los métodos y atributos de un objeto tienen una determinada “visibilidad”, esto es, nuestro objeto puede tener unos métodos que non podamos ejecutar (pero se puede ejecutar por el propio método).
- Un ejemplo puede ser: Una cafetera prepara un café, pero hace una serie de operaciones que tienen que ver con el funcionamiento interno para preparar o café.
- De la misma manera, cuando un objeto ejecuta un método, puede llamar a otros método internos, que no tienen que estar disponibles de cara al exterior

- Existen 4 modificadores de acceso

Modificador	Visibilidad
public	Total
protected	A nivel de paquete
Sen modificador	A nivel de paquete
private	Solo la propia clase

- Los modificadores aplican a métodos o variables

```
public class Playground {

    public static final String MY_CONST = "CONST";
    protected int customInteger = 5;
    double customDouble = 2.3;
    private final float customFloat = 8.63f;

    private int complexOperation() {
        return 2 * 2;
    }

    protected void setCustomDouble(double d) {
        this.customDouble = d;
    }

    public float getCustomFloat() {
        return this.customFloat;
    }

    public static void main(String[] args) {

    }
}
```

- Los paquetes en java son contenedores que almacenan clases, que agrupan las partes de un programa que, de manera general, tienen una funcionalidad o elementos comunes, indicando la ubicación de las clases en una jerarquía de directorios
- Ejemplo de paquetes

```java
package es.imatia.units.playground;

import es.imatia.units.resources.Doctor;
import es.imatia.units.resources.Input;
import es.imatia.units.resources.Person;
import es.imatia.units.resources.PoliceOfficer;
import es.imatia.units.resources.Teacher;

public class Playground {
```

```java
package es.imatia.units.resources;

public class Doctor extends Person{
```

```java
package es.imatia.units.resources;

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Input {
```

```java
package es.imatia.units.resources;

public class Person {
```

```java
package es.imatia.units.resources;

public class Teacher extends Person {
```

```java
package es.imatia.units.resources;

public class PoliceOfficer extends Person{
```

- Ejemplo de clase: Crearemos la clase “Coche”, definiendo primero cuáles son os atributos que tienen todos los coches y cuáles son las acciones (métodos) que pueden hacer todos os coches:

Atributos	Métodos
Marca	Arrancar
Modelo	Apagar
Velocidade máxima	Acelerar
Tipo combustible	Frear
Velocímetro	Xirar o volante
Tacómetro	Dar marcha atrás

```java
public class Car {
    public String brand;
    public String model;
    public static final int MAX_SPEED = 120;
    public String fuel;
    public int speedometer = 0;
    public int tachometer = 0;
    public String gear = "N";
    public boolean reverse = false;
    public int steeringWheelAngle = 0;

    public Car(String brand, String model, String fuel) {
        this.brand = brand;
        this.model = model;
        this.fuel = fuel;
    }

    public static void main(String[] args) {
        Car myCar = new Car("Citroën", "Xsara", "Diésel");
    }
}
```

Clases y objectos: Creación de una clase

```
public class Car {
    public String brand;
    public String model;
    public static final int MAX_SPEED = 120;
    public String fuel;
    public int speedometer = 0;
    public int tachometer = 0;
    public String gear = "N";
    public boolean reverse = false;
    public int steeringWheelAngle = 0;

    public Car(String brand, String model, String fuel) {

    public Car() {

    public void start() {
        if (this.tachometer == 0) {
            this.tachometer = 1000;
            System.out.println("Vehículo acendido");
        } else {
            System.out.println("O vehículo xa está acendido");
        }
    }

    public void stop() {
        if (this.speedometer == 0) {
            this.tachometer = 0;
            System.out.println("Vehículo apagado");
        } else {
            System.out.println("Non se pode apagar o vehículo, primero ten que estar detido");
        }
    }

    public void accelerate() {

    public void brake() {

    public void turnSteeringWheel(int angle) {

    public String showSteeringWheelDetail() {

    public boolean isReverse() {

    public void setReverse(boolean reverse) {

    public void showDetails() {

    public static void main(String[] args) {

}
```

- Además de los atributos, se crean todos los métodos que tiene que tener la clase.
- Estos métodos serán los que el objeto pueda hacer.
- Un operador muy importante es el operador “this”

- Completemos la clase “Coche”

- Los métodos normalmente se invocan desde un objeto, pero existen algunos que son invocados desde la propia clase. De la misma manera, existen atributos de clase que son los mismos para todas las instancias de la mesma clase (esto es, todo los objetos que pertenezcan a mesma clase)

```
public class CustomObject {

    public int actualFuel = 10;
    public static String fuelLabel = "Gasolina";

    public int getActualFuel() {
        return this.actualFuel;
    }

    public void setActualFuel(int actualFuel) {
        if (actualFuel >= 0) {
            this.actualFuel = actualFuel;
        } else {
            System.out.println("A capacidade non pode ser negativa.");
        }
    }

    public static String getFuelLabel() {
        return CustomObject.fuelLabel;
    }

    public static void setFuelLabel(String fuelLabel) {
        CustomObject.fuelLabel = fuelLabel;
    }

    public void showDetails() {
        System.out.println("O tipo de combustible é: " + CustomObject.getFuelLabel());
    }

    public static void main(String[] args) {
        CustomObject cO = new CustomObject();
        cO.showDetails();
        CustomObject.setFuelLabel("Diésel");
        cO.showDetails();
        System.out.println("O tipo de combustible da clase é: " + CustomObject.getFuelLabel());
    }
}
```

- Dado el siguiente código:

```
public class CustomObject {

    public int actualFuel = 10;


    public void showDetails() {
        System.out.println("A capacidade actual é de "
                + this.actualFuel + " litros.");
    }


    public static void main(String[] args) {
        CustomObject cO = new CustomObject();
        cO.showDetails();
        System.out.println("Actualización capacidade");
        cO.actualFuel = -8;
        cO.showDetails();
    }
}
```

- Esta es la salida por consola:

```
<terminated> CustomObject [Java Application] C:\Programs\Java\x64\1.7\jdk1.7.0_75\bin\javaw.exe (11 oct. 2018 13:53:09)
A capacidade actual é de 10 litros.
Actualización capacidade
A capacidade actual é de -8 litros.
```

- ¿Cómo evitar que se establezca una capacidad negativa?

- La encapsulación permite modificar los atributos de una clase mediante métodos, de esta manera, aislamos el acceso a los atributos directamente.

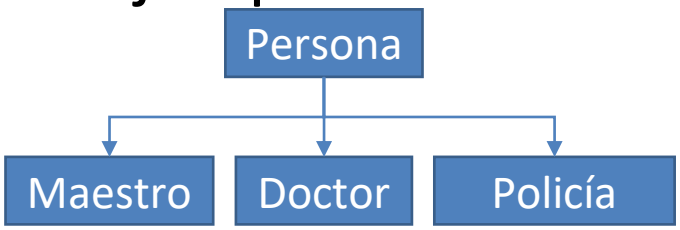
```
public class CustomObject {

    public int actualFuel = 10;

    public int getActualFuel() {
        return this.actualFuel;
    }

    public void setActualFuel(int actualFuel) {
        if (actualFuel >= 0) {
            this.actualFuel = actualFuel;
        } else {
            System.out.println("A capacidade non pode ser negativa.");
        }
    }

    public void showDetails() {
        System.out.println("A capacidade actual é de " + this.getActualFuel() + " litros.");
    }

    public static void main(String[] args) {
        CustomObject cO = new CustomObject();
        cO.showDetails();
        System.out.println("Actualización capacidade");
        cO.setActualFuel(-8);
        cO.showDetails();
    }
}
```

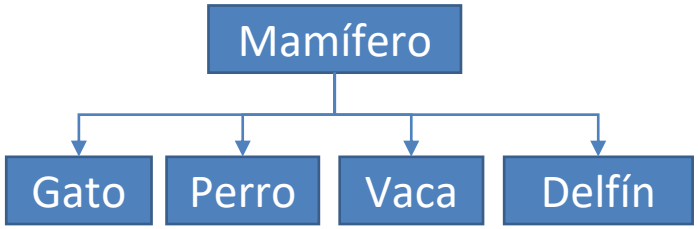
- Estos métodos se conocen como *getters* y *setters*, e la regla común para nombrarlos es anteponiendo la palabra *set* para establecer valores y *get* para recuperarlos (se antepone *is* para recuperar el valor de booleanos)

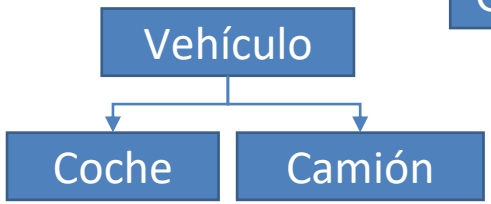
- Lo normal cuando se usa la encapsulación es marcar todos los atributos como *protected* o *private*, y tener getter e setters públicos para modificarlos
- Para obtener un buen diseño de un programa, los atributos que son públicos tienen que ser constantes inmutables, escribiéndose como *public static final*:

```
public static final String NOT_HYBRID = "NON HÍBRIDO";
```

- A herencia es el sistemas de la POO que permite reutilizar clases, añadiendo funcionalidades nuevas y específicas a clases que ya existen, o cambiando el comportamiento de algunas de ellas.
- La herencia permite que el contenido de la clase “padre” (superclase) esté contenido en la clase “hija” (subclase)
- Ejemplos de herencia:





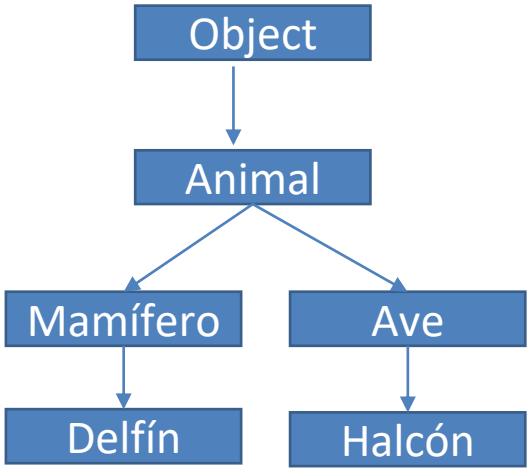


- En la orientación a objetos, las clases Boeing, Harrier, Airbus son subclases de Avión
- Las subclases indican una ***especialización*** de la clase base y a su vez, la superclase es una ***generalización***.
- Para seguir el paradigma de la orientación a objectos, cada subclase tiene que ser una especialización de la clase base
- Todas las acciones referentes as ***generalidades*** de una subclase se tienen que ejecutar en la clase base

- La extensión de la clase se indica co la palabra *extends*.
- Todas las clases forman parte de una jerarquía de clases. La clase de la que derivan todas las demás se llama *Object*
- Dado como ejemplo las clases Animal, Mamífero, Ave, Delfín y Halcón, solo se indica explícitamente la extensión a partir de Animal y es en esta donde la extensión de Object é implícita
- Las clases heredan los métodos de la superclase Object

```
public class Animal {
    ...
}

public class Mammal extends Animal{
    ...
}
```

```
public class Bird extends Animal{
    ...
}
public class Dolphin extends Mammal{
    ...
}
public class Falcon extends Bird{
    ...
}
```



- Ejemplo

```java
public class Person {
    protected String name;
    protected String surname;

    public Person(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }

    public void getDetails() {
        System.out.println("Nome completo: "+ name +" "+ surname);
    }
}
```

```java
public class PoliceOfficer extends Person{

    protected String squad;

    public PoliceOfficer(String name, String surname, String squad) {
        super(name, surname);
        this.squad = squad;
    }
}
```

```java
public class Teacher extends Person {

    protected String area;

    public Teacher(String name, String surname, String area) {
        super(name, surname);
        this.area = area;
    }
}
```

```java
public class Doctor extends Person{

    protected String specialization;

    public Doctor(String name, String surname, String specialization) {
        super(name, surname);
        this.specialization = specialization;
    }
}
```

- Ejemplo:

```
public class Playground {

    public static void main(String[] args) {

        Person p = new Person("John", "Smith");
        Teacher t = new Teacher("María", "Montessori", "Educación");
        PoliceOfficer po = new PoliceOfficer("Jake", "Peralta", "B-99");
        Doctor d = new Doctor("Gregory", "House", "Nefroloxía e infectoloxía");

        p.getDetails();
        t.getDetails();
        po.getDetails();
        d.getDetails();
    }
}
```

```
<terminated> Playground [Java Application] C:\Programs\Java\x64\1.7\jdk1.7.0_75\bin\javaw.exe
Nome completo: John Smith
Nome completo: María Montessori
Nome completo: Jake Peralta
Nome completo: Gregory House
```

- Aunque no se definió el método *getDetails()* para las clases *Teacher*, *PoliceOfficer* y *Doctor*, este método está definido en la clase base, por lo que las subclases pueden usarlo.

- La accesibilidad de los métodos de la clase padre viene dada según sus modificadores.

Modificador	Accesibilidad			
	La misma clase	Mismo paquete	Subclase	Otro paquete
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
Sin modificador	Sí	Sí	No	No
private	Sí	No	No	No

- La firma de un método es la combinación del nombre del método y los argumentos recibidos.
- La sobrecarga permite múltiples métodos con el mismo nombre pero con diferentes argumentos
- También existe la sobrecarga de constructores

```
public void showDetailForceVector(int xF) {
    int xFinal = xF - this.getRandomNumber(-10, 10);
    int yFinal = 0 - this.getRandomNumber(-10, 10);
    int zFinal = 0 - this.getRandomNumber(-10, 10);
    System.out.println("El vector recultante es: " + xFinal + ", " + yFinal + ", " + zFinal);
}

public void showDetailForceVector(int xF, int yF) {
    int xFinal = xF - this.getRandomNumber(-10, 10);
    int yFinal = yF - this.getRandomNumber(-10, 10);
    int zFinal = 0 - this.getRandomNumber(-10, 10);
    System.out.println("El vector recultante es: " + xFinal + ", " + yFinal + ", " + zFinal);
}

public void showDetailForceVector(int xF, int yF, int zF) {
    int xFinal = xF - this.getRandomNumber(-10, 10);
    int yFinal = yF - this.getRandomNumber(-10, 10);
    int zFinal = zF - this.getRandomNumber(-10, 10);
    System.out.println("El vector recultante es: " + xFinal + ", " + yFinal + ", " + zFinal);
}
```

```
public TVRemote() {
    this.channel = 0;
    this.volume = 20;
    this.color = "Negro";
}

public TVRemote(String color) {
    this.channel = 0;
    this.volume = 20;
    this.color = color;
}
```

- La sobreescritura de un método permite que se modifique la acción de la clase base en la clase derivada

```java
public class Person {
    protected String name;
    protected String surname;

    public Person(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }

    public void getDetails() {
        System.out.println("Nome completo: "+ name +" "+ surname);
    }
}
```

```java
public class Doctor extends Person {

    protected String specialization;

    public Doctor(String name, String surname, String specialization) {
        super(name, surname);
        this.specialization = specialization;
    }

    @Override
    public void getDetails() {
        System.out.println("Doctor " + name + " " + surname
                + ", especialista en " + specialization.toLowerCase());
    }
}
```

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

    Person p = new Person("John", "Smith");
    Teacher t = new Teacher("María", "Montessori", "Educación");
    PoliceOfficer po = new PoliceOfficer("Jake", "Peralta", "B-99");
    Doctor d = new Doctor("Gregory", "House", "Nefroloxía e infectoloxía");

    p.getDetails();
    t.getDetails();
    po.getDetails();
    d.getDetails();
}
```

- Es recomendable el uso de la anotación *@Override*

```
Nome completo: John Smith
Nome completo: María Montessori
Nome completo: Jake Peralta
Doctor Gregory House, especialista en nefroloxía e infectoloxía
```

- Las clases abstractas son similares a las clases normales, aunque tienen una característica especial, dichas clases abstractas no puoden instanciar objetos.
- La clase abstracta tiene la misma estructura que una clase normal, pero precedida de la palabra clave ***abstract*** al inicio de su declaración
- Las clases abstractas pueden tener métodos abstractos, que son métodos que no tienen implementación en la clase abstracta, pero tendrán implementación en una clase concreta.
- Si una clase tiene un método abstracto, tiene que ser necesariamente una clase abstracta

- Ejemplo de clase abstracta

```
public abstract class Merchandise {

    protected String name;
    protected String uniqueId;
    protected String resposibleId;
    protected int zone;
    protected String area;
    protected String shelf;
    protected int quantity;

    public Merchandise(String name, String uniqueId, String responsibleId) {
        this.name = name;
        this.uniqueId = uniqueId;
        this.resposibleId = responsibleId;
    }

    public Merchandise(String name, String uniqueId, String responsibleId,
            int zone, String area, String shelf, int quantity) {
        this.name = name;
        this.uniqueId = uniqueId;
        this.resposibleId = responsibleId;
        this.zone = zone;
        this.area = area;
        this.shelf = shelf;
        this.quantity = quantity;
    }

    public String getLocation() {
        StringBuilder builder = new StringBuilder();
        builder.append("Z - ");
        builder.append(getZone());
        builder.append(" A - ");
        builder.append(getArea());
        builder.append(" E - ");
        builder.append(getShelf());
        return builder.toString();
    }

    public abstract Object getSpecificData();

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getUniqueId() {
        return uniqueId;
    }

    public String getResposibleId() {
        return resposibleId;
    }

    public void setResposibleId(String resposibleId) {
        this.resposibleId = resposibleId;
    }

    public int getZone() {
        return zone;
    }

    public void setZone(int zone) {
        this.zone = zone;
    }

    public String getArea() {
        return area;
    }

    public void setArea(String area) {
        this.area = area;
    }

    public String getShelf() {
        return shelf;
    }

    public void setShelf(String shelf) {
        this.shelf = shelf;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
}
```

- Ejemplo de clase concreta que extiende de una abstracta

```java
public class FreshMerchandise extends Merchandise {

    protected Date expirationDate;
    protected SimpleDateFormat sdf = new SimpleDateFormat();

    public FreshMerchandise(String name, String uniqueId, String responsibleId) {
        super(name, uniqueId, responsibleId);
        // TODO Auto-generated constructor stub
    }

    public FreshMerchandise(String name, String uniqueId, String responsibleId,
            int zone, String area, String shelf, int quantity, Date expirationDate) {
        super(name, uniqueId, responsibleId, zone, area, shelf, quantity);
        this.expirationDate = expirationDate;
    }

    @Override
    public Object getSpecificData() {
        StringBuilder builder = new StringBuilder();
        builder.append("Localización: ");
        builder.append(getLocation());
        builder.append("\n");
        builder.append("Caducidade: ");
        builder.append(sdf.format(getExpirationDate()));
        return builder.toString();
    }

    public void printSpecificData() {
        System.out.println(getSpecificData());
    }

    public Date getExpirationDate() {
        return expirationDate;
    }

    public void setExpirationDate(Date expirationDate) {
        this.expirationDate = expirationDate;
    }

}
```

```java
FreshMerchandise fm = new FreshMerchandise("Mazás", "001-9",
        "Froitería  de onte S.L.", 8, "C", "114D",
        53, Calendar.getInstance().getTime());
fm.printSpecificData();
```

```
Localización: Z8/AC/E114D
Caducidade: 18/10/18 12:32
```

- La clase final es una clase que no puede ser extendida. De la misma manera, un método final no puede ser sobrescrito

```
public final class FreshMerchandise extends Merchandise {

    protected Date expirationDate;
    protected SimpleDateFormat sdf = new SimpleDateFormat();

    public FreshMerchandise(String name, String uniqueId, String responsibleId) {
        super(name, uniqueId, responsibleId);
        // TODO Auto-generated constructor stub
    }

    public FreshMerchandise(String name, String uniqueId, String responsibleId,
            int zone, String area, String shelf, int quantity, Date expirationDate) {
        super(name, uniqueId, responsibleId, zone, area, shelf, quantity);
        this.expirationDate = expirationDate;
    }

    @Override
    public Object getSpecificData() {
        StringBuilder builder = new StringBuilder();
        builder.append("Localización: ");
        builder.append(getLocation());
        builder.append("\n");
        builder.append("Caducidade: ");
        builder.append(sdf.format(getExpirationDate()));
        return builder.toString();
    }

    public final void printSpecificData() {
        System.out.println(getSpecificData());
    }

    public Date getExpirationDate() {
        return expirationDate;
    }

    public void setExpirationDate(Date expirationDate) {
        this.expirationDate = expirationDate;
    }
}
```

- Una interfaz no es más que una clase que indica qué acciones es obligatorio que pueda ejecutar un determinado objecto de una clase.
- Los métodos indican una acción y una implementación concreta de la acción, la interfaz solo muestra la acción.
- Por ejemplo, la interfaz máquina:

```
public interface IMachine {

    public void start();

    public void stop();

    public void maintenance();

}
```

- Todas las clases que implementen la interfaz *IMachine*, tienen que tener métodos que implementen las acciones de encender, parar y mantenimiento.

- Ejemplos

```java
public class Plane implements IMachine {

    private final String name;

    public Plane(String name) {
        this.name = name;
    }

    @Override
    public void start() {
        System.out.println("Avión acendido");
    }

    @Override
    public void stop() {
        System.out.println("Avión apagado");
    }

    @Override
    public void maintenance() {
        System.out.println("Realizando o mantemento do avión");
    }

    public void takeOff() {
        System.out.println("O avión despega");
    }

    public void land() {
        System.out.println("O avión aterriza");
    }

    public void fly() {
        System.out.println("O avión está a voar");
    }
}
```

```java
public class Tractor implements IMachine{

    protected int horsePower = 0;

    public Tractor(int hp) {
        this.horsePower = hp;
    }

    @Override
    public void start() {
        System.out.println("O tractor está acendido.");
    }

    @Override
    public void stop() {
        System.out.println("O tractor está apagado.");
    }

    @Override
    public void maintenance() {
        System.out.println("O tractor está en mantemento.");
    }

    public void forward() {
        System.out.println("O tractor avanza");
    }

    public void backward() {
        System.out.println("O tractor retrocede");
    }
}
```

- Diferencia y uso de las clases abstractas y las interfaces
- Las clases abstractas sirven para utilizar un esqueleto de funcionalidades base de nuestra clase, pero que habrá ciertas partes, que por deseño, no se puedan equiparar. De esta forma, solo se podrán instanciarse objectos de una clase derivada.
- Las interfaces muestran el conjunto de acciones que pueden hacer esos objetos instanciados

- El polimorfismo es la capacidad, dentro de una relación de herencia, de que cualquier objeto de la superclase puede almacenar un objecto de cualquiera de sus subclases
- La clase padre puede almacenar las clases derivadas, pero no puede darse el caso inverso.

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

    Person p = new Person("John", "Smith");
    Person t = new Teacher("María", "Montessori", "Educación");
    Person po = new PoliceOfficer("Jake", "Peralta", "B-99");
    Person d = new Doctor("Gregory", "House", "Nefroloxía e infectoloxía");

    p.getDetails();
    t.getDetails();
    po.getDetails();
    d.getDetails();

}
```

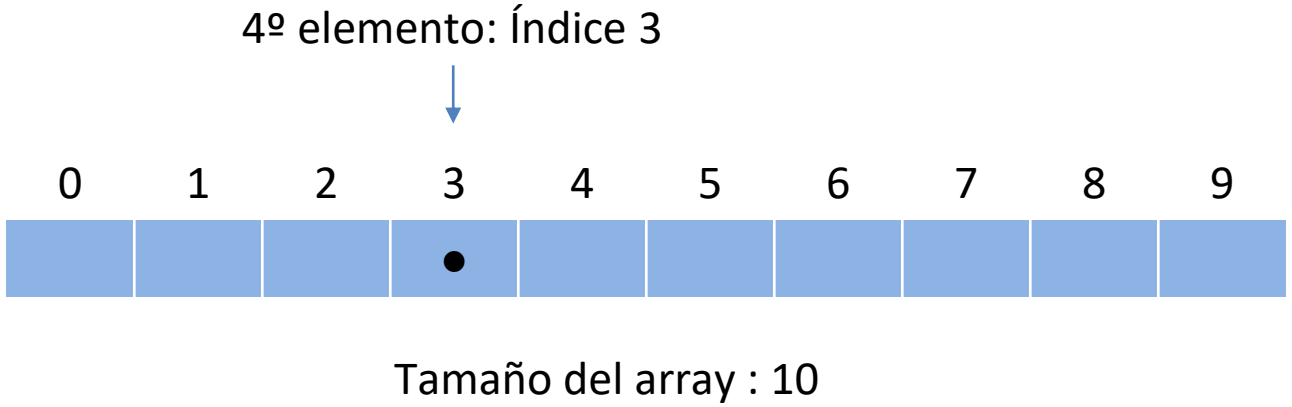
```
Nome completo: John Smith
Nome completo: María Montessori
Nome completo: Jake Peralta
Doctor Gregory House, especialista en nefroloxía e infectoloxía
```

- De la misma forma, se puede emplear el polimorfismo con las interfaces

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

    IMachine plane = new Plane("Boing");
    IMachine tractor = new Tractor(3500);

    plane.start();
    tractor.start();

    ((Plane)plane).fly();
    ((Tractor)tractor).forward();
}
```

```
Avión acendido
O tractor está acendido.
O avión está a voar
O tractor avanza
```

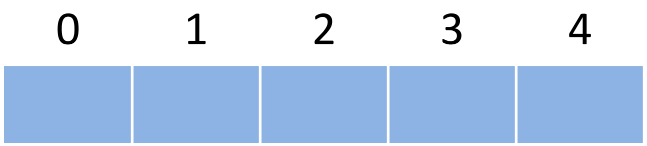
- Una estructura de almacenamiento sirve para almacenar múltiples datos del mismo tipo.
- Un array es un conjunto de datos almacenados de forma secuencial en memoria
- Se accede a los datos mediante un índice, empezando este índice en 0



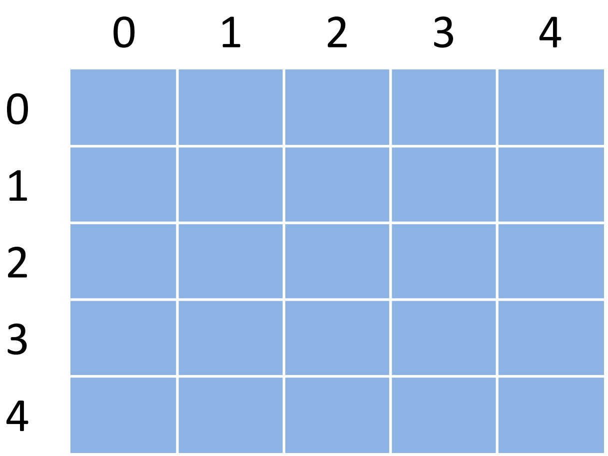
- En la declaración de un array se establece el tamaño, o se deduce según la inicialización empleada.

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

    //Inicialización dos arrays

    int [] intArray = new int[3];
    String [] stringArray = { "1", "2", "3" };

    // Establecer datos nun array

    intArray[0] = 10;
    intArray[1] = 9;
    intArray[2] = 8;


    // Mostar datos do array dunha posición

    System.out.println(intArray[0]);
    System.out.println(stringArray[0]);

    // Capacidade do array

    System.out.println(intArray.length);
    System.out.println(stringArray.length);
}
```

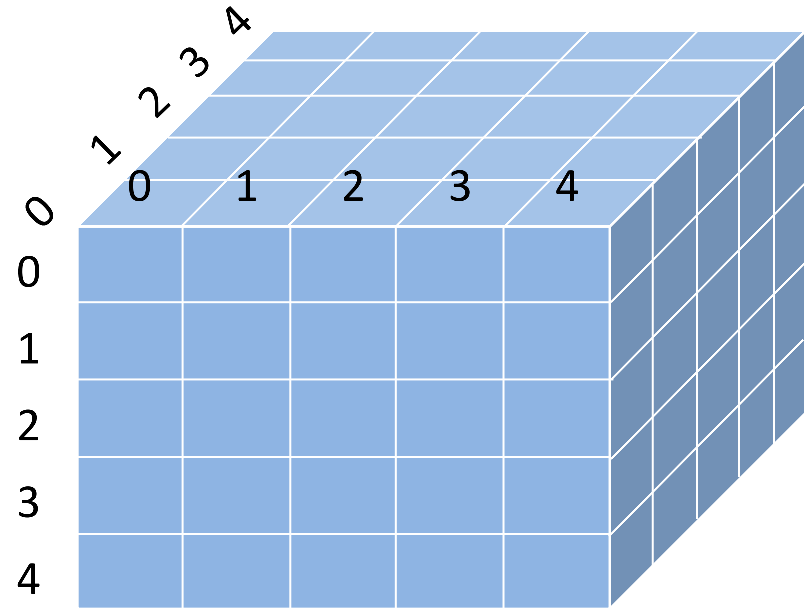
- Los arrays pueden ser multidimensionales




```
int [] intArrayUni = {1, 2, 3, 4, 5};
```




```
int [][] intArrayBi = {
        {1, 2, 3, 4, 5},
        {1, 2, 3, 4, 5},
        {1, 2, 3, 4, 5},
        {1, 2, 3, 4, 5},
        {1, 2, 3, 4, 5}
        };
```




```
int [][][] intArrayTri = {
    {{1, 1, 1, 1, 1},{2, 2, 2, 2, 2},{3, 3, 3, 3, 3},{4, 4, 4, 4, 4},{5, 5, 5, 5, 5}},
    {{1, 1, 1, 1, 1},{2, 2, 2, 2, 2},{3, 3, 3, 3, 3},{4, 4, 4, 4, 4},{5, 5, 5, 5, 5}},
    {{1, 1, 1, 1, 1},{2, 2, 2, 2, 2},{3, 3, 3, 3, 3},{4, 4, 4, 4, 4},{5, 5, 5, 5, 5}},
    {{1, 1, 1, 1, 1},{2, 2, 2, 2, 2},{3, 3, 3, 3, 3},{4, 4, 4, 4, 4},{5, 5, 5, 5, 5}},
    {{1, 1, 1, 1, 1},{2, 2, 2, 2, 2},{3, 3, 3, 3, 3},{4, 4, 4, 4, 4},{5, 5, 5, 5, 5}}
    };
```

- Para recorrer un array, podemos usar un bucle for:

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

    //Inicialización dos arrays

    int [] intArray = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    // Recorrido do array

    for (int i = 0; i < intArray.length; i++){
        System.out.print(intArray[i]+ " ");
    }
}
```

```
1 2 3 4 5 6 7 8 9 10
```

- Para recorrer un array, podemos usar un bucle for:

```
System.out.println("Unidimensional");
for (int i = 0; i < intArrayUni.length; i++){
    System.out.print(intArrayUni[i]+ " ");
}

System.out.println("\n\nBidimensional");
for (int i = 0; i < 5; i++){
    for (int j = 0; j < 5; j++){
        System.out.print(intArrayBi[i][j]+ " ");
    }
    System.out.println();
}

System.out.println("\nSuma Tridimensional");
for (int i = 0; i < 5; i++){
    for (int j = 0; j < 5; j++){
        int val = 0;
        for (int k = 0; k < 5; k++){
            val+= intArrayTri[i][j][k];
        }
        System.out.print(val+ " ");
    }
    System.out.println();
}
```

```
Unidimensional
1 2 3 4 5
```

```
Bidimensional
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

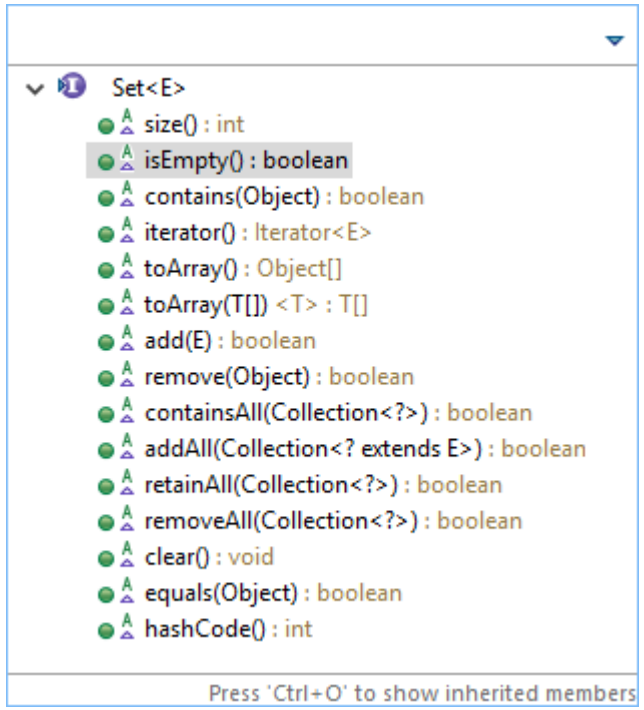
```
Suma Tridimensional
5 10 15 20 25
5 10 15 20 25
5 10 15 20 25
5 10 15 20 25
5 10 15 20 25
```

- Las colecciones sirven para almacenar datos (son estructuras de almacenamiento).
- Las colecciones extienden la interfaz *Collection<E>*, y según su implementación, permiten almacenar y recorrer la estructura de diferentes maneras.

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

    List<Person> stringList = new ArrayList<>();

    stringList.add(new Person("John", "Smith"));
    stringList.add(new Teacher("María", "Montessori", "Educación"));
    stringList.add(new PoliceOfficer("Jake", "Peralta", "B-99"));
    stringList.add(new Doctor("Gregory", "House", "Nefroloxía e infectoloxía"));

    for (Person p: stringList) {
        p.getDetails();
    }

}
```

- Las colecciones más comunes son:
  - Conjuntos
  - Listas
  - Mapas
  - Colas
- Los conjuntos no admiten dos elementos iguales
- Las listas permiten múltiples elementos repetidos, e respeta el orden
- Los mapas son colecciones que asocian una clave a un valor. La clave non puede ser asociada a múltiples valores
- Las colas solo pueden manejar os objetos del principio o final, dependiendo la implementación

- Conjuntos
- Estos son os métodos que tienen los conjuntos
- Los conjuntos más comunes:
  - Hashset
  - TreeSet



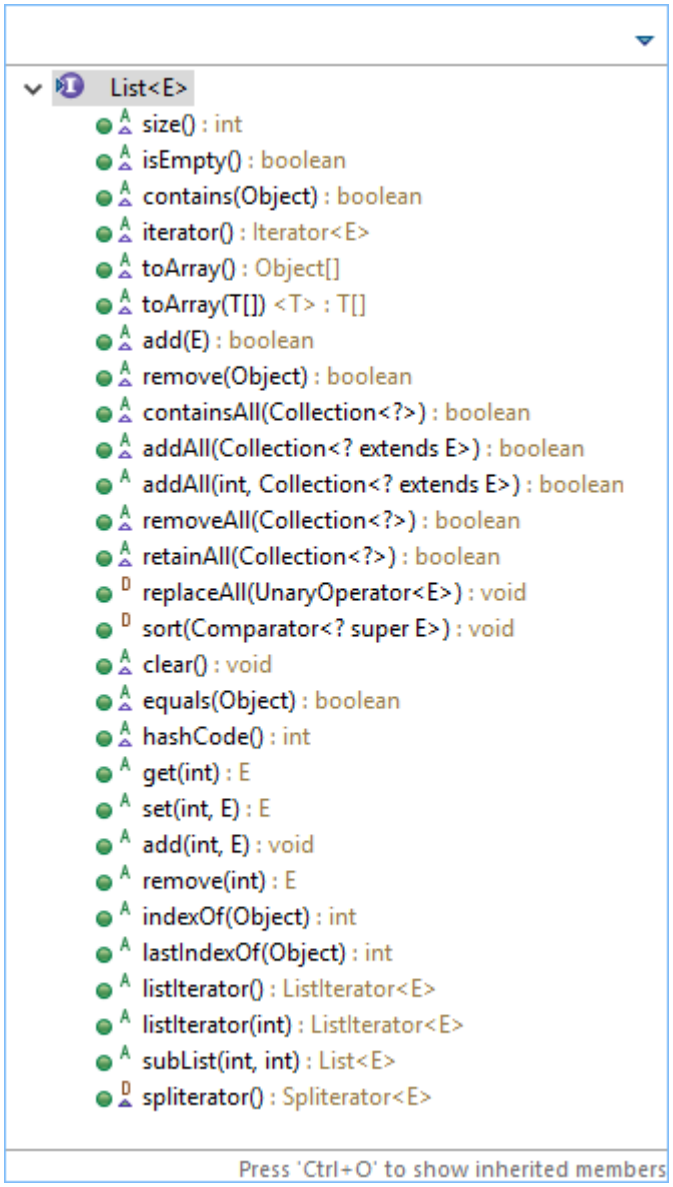

```
public static void main(String[] args) {

    Set<String> customSet = new HashSet<>();
    customSet.add("Libreta");
    customSet.add("Bolígrafo");
    customSet.add("Lápiz");
    customSet.remove("Bolígrafo");

    for (String s: customSet) {
        System.out.println(s);
    }
}
```

```
Libreta
Lápiz
```

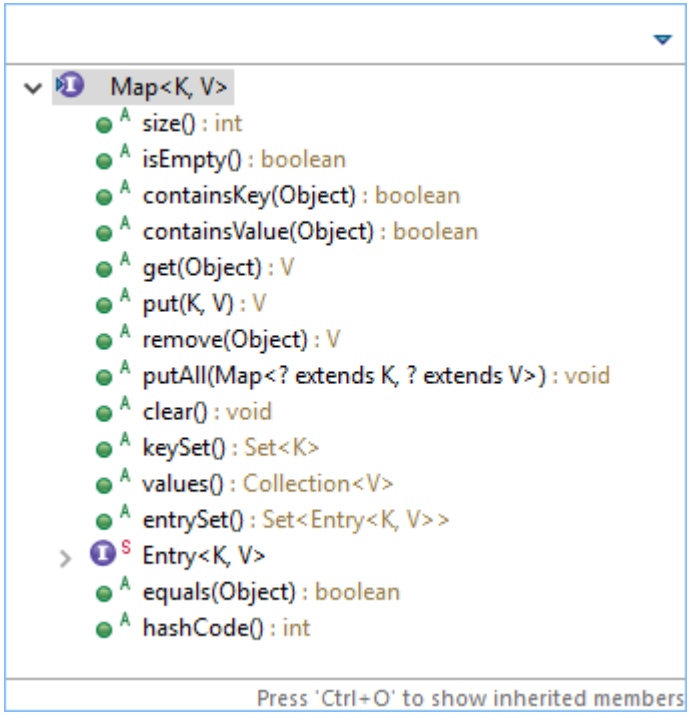
- Listas
- Estos son los métodos que tienen las listas
- Las listas más comunes:
  - ArrayList
  - LinkedList

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

    List<String> customList = new ArrayList<>();
    customList.add("Libreta");
    customList.add("Bolígrafo");
    customList.add("Lápiz");

    for (String s: customList) {
        System.out.println(s + " está na posición " + customList.indexOf(s) );
    }
}
```

```
Libreta está na posición 0
Bolígrafo está na posición 1
Lápiz está na posición 2
```



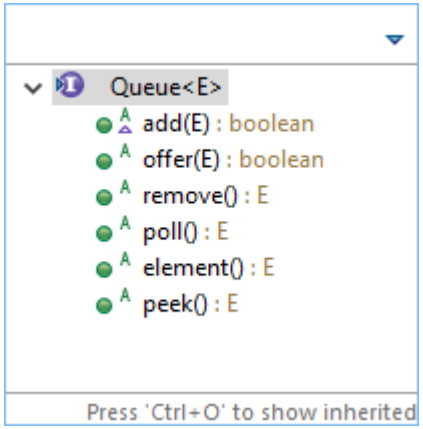
- Mapas
- Estos son los métodos que tienen los mapas
- Los mapas más comunes:
  - HashMap
  - LinkedHashMap




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

    Map<String,Person> customMap = new HashMap<>();

    customMap.put("person", new Person("John", "Smith"));
    customMap.put("teacher", new Teacher("María", "Montessori", "Educación"));
    customMap.put("police", new PoliceOfficer("Jake", "Peralta", "B-99"));
    customMap.put("doctor", new Doctor("Gregory", "House", "Nefroloxía e infectoloxía"));

    customMap.get("teacher").getDetails();
    customMap.remove("teacher");

    System.out.println("Contén a clave \"police\": " + customMap.containsKey("police"));
    System.out.println("Contén a clave \"teacher\": " + customMap.containsKey("teacher"));
}
```

```
Nome completo: María Montessori
Contén a clave "police": true
Contén a clave "teacher": false
```

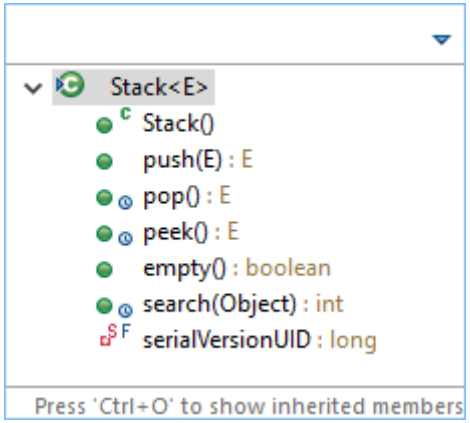
- Colas
- Las colas son listas en las que los elementos se introducen y eliminan por diferentes extremos
- FIFO -> First In - First Out




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

    Queue<String> customQueue= new LinkedList<>();

    customQueue.offer("Smith");
    customQueue.offer("Montessori");
    customQueue.offer("Peralta");
    customQueue.offer("House");

    System.out.println(customQueue.peek()+"\n");

    while (!customQueue.isEmpty()) {
        System.out.println(customQueue.poll());
    }
}
```

```
Smith

Smith
Montessori
Peralta
House
```

- Pilas
- As pilas son una colección de elementos que se introducen y eliminan por el mesmo extremo
- LIFO -> Last In – First Out




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

    Stack<String> customQueue = new Stack<>();

    customQueue.push("Smith");
    customQueue.push("Montessori");
    customQueue.push("Peralta");
    customQueue.push("House");

    System.out.println(customQueue.peek()+"\n");

    while (!customQueue.isEmpty()) {
        System.out.println(customQueue.pop());
    }
}
```

```
House

House
Peralta
Montessori
Smith
```

- El lenguaje XML es un lenguaje de marcado extensible *(e**X**tensible **M**arkup **L**anguage)*.
- Es legible tanto por humanos como por máquinas
- Independiente de la plataforma
- Sirve para intercambiar información entre sistemas

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<component>
    <components>
        <component quantity="1">Tarxéta gráfica</component>
        <component quantity="1">CPU</component>
        <component quantity="3">Ventilador</component>
        <component quantity="1">Placa base</component>
        <component quantity="2">Memoria RAM</component>
        <component quantity="2">Disco duro</component>
        <component quantity="1">Fonte de alimentación</component>
        <component quantity="1">Caixa</component>
    </components>
</component>
```

- JSON es un formato ligeiro de intercambio de datos
- **J**ava**S**cript **O**bject **N**otation
- Es simple de leer por las personas y de interpretarse y generarse por máquinas
- Formado por pares clave-valor
- Puede utilizarse independiente del lenguaje Javascript


```
{
  "component": {
    "components": {
      "component": [
        {
          "-quantity": "1",
          "#text": "Tarxéta gráfica"
        },
        {
          "-quantity": "1",
          "#text": "CPU"
        },
        {
          "-quantity": "3",
          "#text": "Ventilador"
        },
        {
          "-quantity": "1",
          "#text": "Placa base"
        },
        {
          "-quantity": "2",
          "#text": "Memoria RAM"
        },
        {
          "-quantity": "2",
          "#text": "Disco duro"
        },
        {
          "-quantity": "1",
          "#text": "Fonte de alimentación"
        },
        {
          "-quantity": "1",
          "#text": "Caixa"
        }
      ]
    }
  }
}
```

- En un programa de Java puede existir algún tipo de problema (error) durante la ejecución del mismo.
- Cando esto sucede, se lanza una excepción. Puede ser por dividir entre 0, disco duro lleno, intentar acceder a una posición de un vector que no existe, hacer un cast inválido….

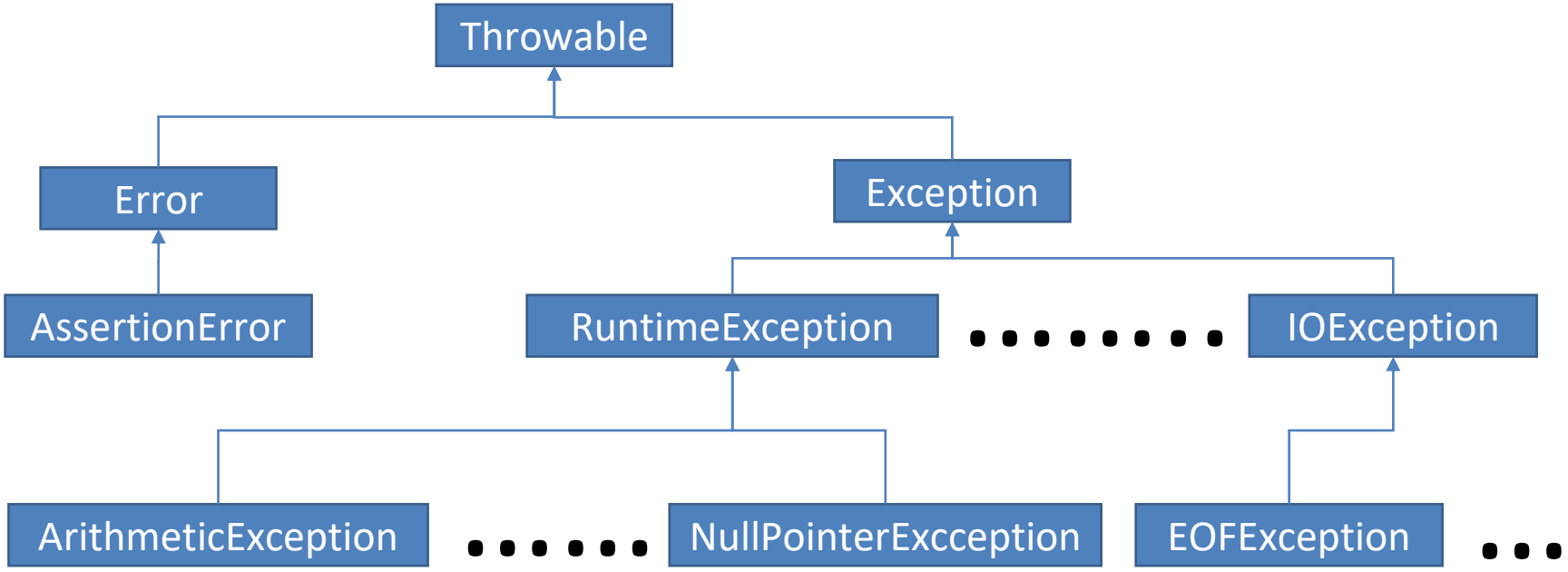
```
public static void main(String[] args) {

    int dividendo = 3, divisor = 0;
    int res = dividendo / divisor;
    System.out.println(res);

}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at es.imatia.units.playground.Playground.main(Playground.java:8)
```

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

    int num = Input.integer("Introduce un número: ");
    System.out.println(num);

}
```

```
Introduce un número: boas!
Exception in thread "main" java.lang.NumberFormatException: For input string: "boas!"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:492)
        at java.lang.Integer.parseInt(Integer.java:527)
        at es.imatia.units.resources.Input.integer(Input.java:27)
        at es.imatia.units.playground.Playground.main(Playground.java:9)
```

- Cuando una excepción se lanza, la ejecución del programa no continúa
- A jerarquía de las excepciones es la siguiente:



- Para controlar una excepción y continuar la ejecución:
  - Try-catch-finally

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

    int dividendo = 3, divisor = 0;
    try {
        int res = dividendo / divisor;
        System.out.println(res);
    } catch (ArithmeticException e) {
        System.out.println("Non se pode dividir por 0");
    } finally {
        System.out.println("Programa acabado");
    }
}
```

```
Non se pode dividir por 0
Programa acabado
```

- Para indicar que se puede lanzar una excepción:
  - Throws

```
public class Playground {

    public void readFile() throws IOException {
        File file = new File("file.txt");
        BufferedReader br = new BufferedReader(new FileReader(file));
        String readLine = "";

        System.out.println("Contido do ficheiro:\n");

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

    public static void main(String[] args) {

        Playground p = new Playground();

        try {
            p.readFile();
        } catch (IOException e) {
            System.out.println("Produciuse unha excepción!");
            e.printStackTrace();
        } finally {
            System.out.println("\nFin do programa");
        }

    }
}
```

```
Produciuse unha excepción!
java.io.FileNotFoundException: file.txt (El sistema no puede encontrar el archivo especificado)

Fin do programa
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:146)
        at java.io.FileReader.<init>(FileReader.java:72)
        at es.imatia.units.playground.Playground.readFile(Playground.java:13)
        at es.imatia.units.playground.Playground.main(Playground.java:28)
```

- Las excepciones se pueden extender para crear excepciones propias y lanzarlas

```java
public class Playground {

    public int division(int dividendo, int divisor) throws ByZeroException {
        if (divisor == 0) {
            throw new ByZeroException("Non se pode dividir un número entre 0");
        } else {
            return dividendo / divisor;
        }

    }

    public static void main(String[] args) {
        Playground p = new Playground();
        try {
            System.out.println(p.division(10, 0));
        } catch (ByZeroException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("Programa acabado.");
    }
}
```

```java
public class ByZeroException extends ArithmeticException{

    public ByZeroException() {
        super();
    }

    public ByZeroException(String s) {
        super(s);
    }
}
```

- En un programa es muy común que se necesite almacenar la información generada.
- Java permite esa acción mediante el uso de *streams* o *flujos*, una abstracción de todo lo que produzca o consuma información.
- Java provee dos tipos de *streams*:
  - Flujo de bytes: Orientados al manejo de lectura o escritura de datos binarios
  - Flujo de caracteres: Orientados al manejo de lectura o escritura de caracteres. Internamente, se transforma en un flujo de bytes

- La clase Input que utilizamos recoje información del teclado *(System.in)*
- Para escribir en la consola, utilizamos el método *System.out.print()* ou *System.out.println()*

```
public static String init() {
    String buffer = "";
    InputStreamReader stream = new InputStreamReader(System.in);
    BufferedReader reader = new BufferedReader(stream);
    try {
        buffer = reader.readLine();
    } catch (Exception e) {
        System.out.append("Dato non válido.");
    }
    return buffer;
}
```

```
public static int integer(String message) {
    if (message != null) {
        System.out.print(message);
    }
    int value = Integer.parseInt(Input.init());
    return value;
}
```

- La lectura y escritura de ficheros

```
public static void main(String[] args) {
    BufferedReader br = null;

    try {
        br = new BufferedReader(new FileReader(new File("file.txt")));
        String linea = "";
        while ((linea = br.readLine()) != null) {
            System.out.println(linea);
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        try {
            if (null != br) {
                br.close();
            }
        } catch (Exception e2) {
            e2.printStackTrace();
        }
    }
}
```

- La lectura y escritura de ficheros

```
public static void main(String[] args) {
    File file = new File("files.txt");
    PrintWriter pw = null;
    try {
        pw = new PrintWriter(new FileWriter(file));

        for (int i = 0; i < 10; i++) {
            pw.println("Línea " + i);
        }

    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        try {
            if (file != null) {
                pw.close();
            }
        } catch (Exception e2) {
            e2.printStackTrace();
        }
    }
}
```

- Para los ficheros binarios se hace de la misma manera, pero, en vez de usar los *Reader* o *Writer*, se usarán los *InputStream* u *OutputStream*.
- En lugar de los métodos *readLine() y println()*, se usarán los métodos *read()* e *write()*
- **FileInputStream**, **FileOuputStream**, **FileReader** o **FileWriter**, hacen múltiples llamadas al disco, por lo que es mejor utilizar los **BufferedReader**, **BufferedInputStream**, **BufferedWriter** y **BufferedOutputStream**, de manera intermedia, para minimizar estos accesos.

- Para poder eliminar un archivo.

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

    File file = new File("files.txt");
    boolean check = file.delete();
    if (check) {
        System.out.println("Borrouse correctamente");
    } else {
        System.out.println("Non se eliminou o arquivo");
    }
}
```