



imatia
innovation

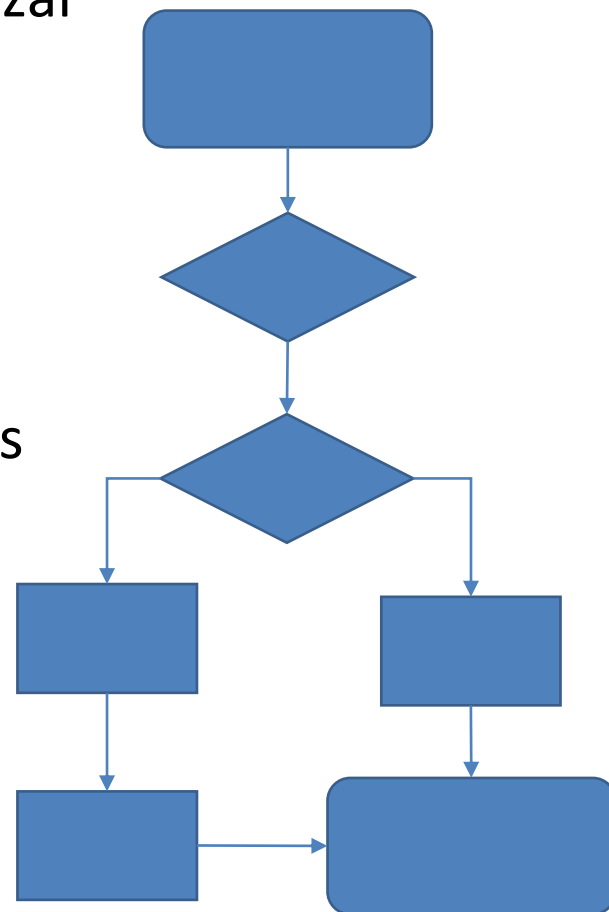
We help you to do more



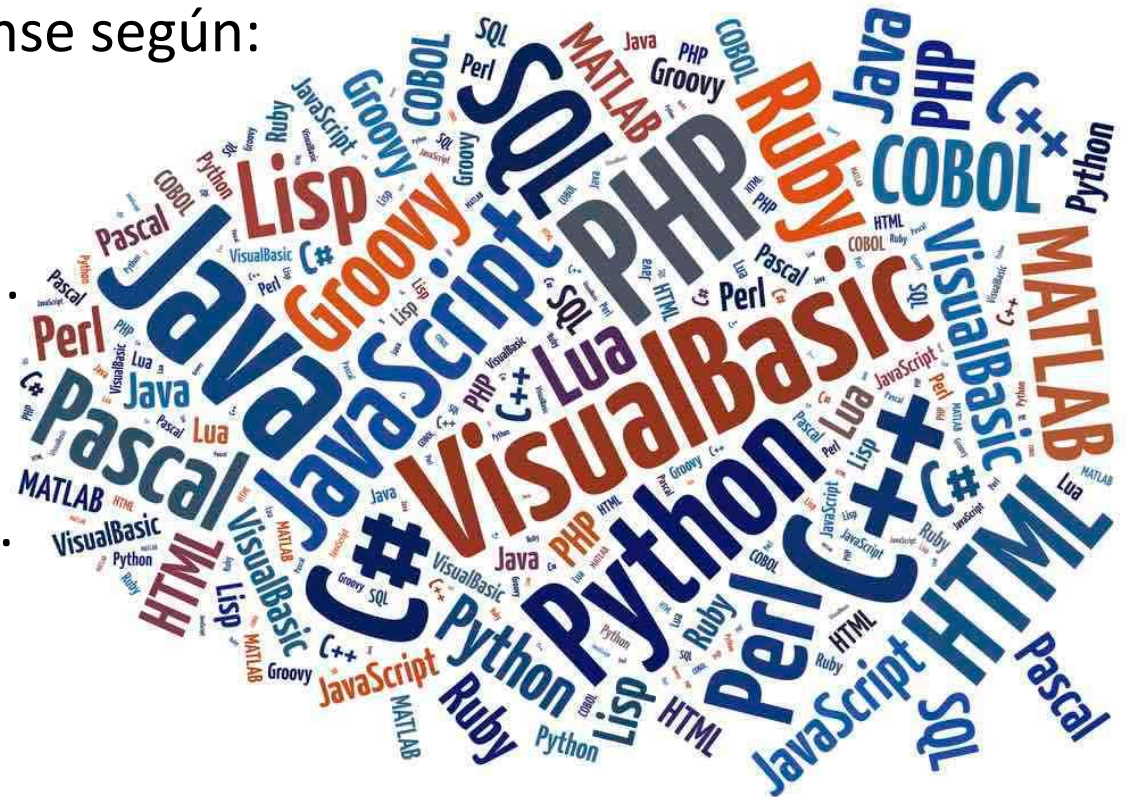
- Introducción a programación.
- Tipos, variables, constantes, asignacións.
- Funcións, procedementos e métodos
- Definición de clase, obxeto e interfaz.
- Estruturas de control.
- Estruturas de almacenamento.
- Lectura e escritura de información.



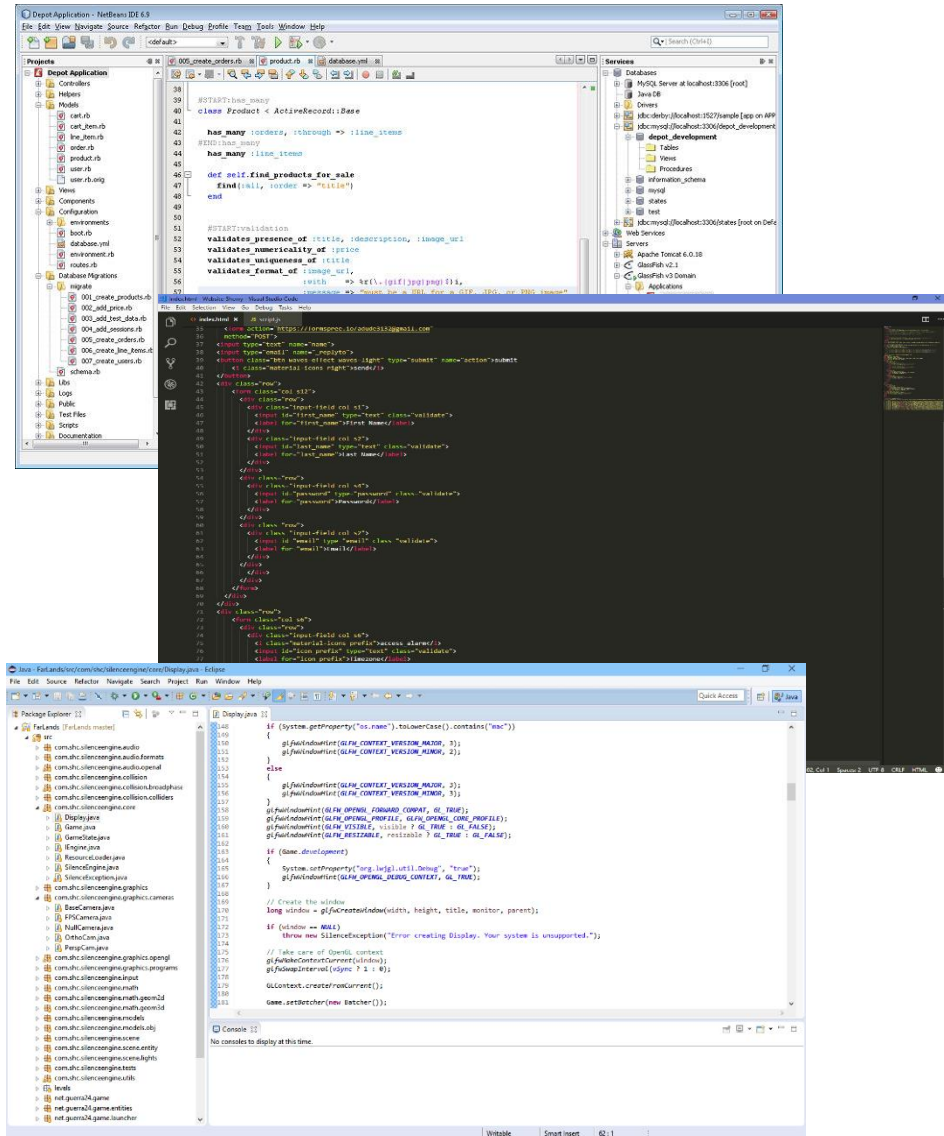
- A **programación** defínese como a *implementación dun algoritmo*, para realizar un **programa**.
- Un **algoritmo** é unha secuencia *non ambigua, finita e ordeada* de instrucións que úsase para resolver un problema.
- A **programación** ten como obxectivo a creación de **programas**, cuxas instrucións un ordenador *é capaz de interpretar e de executar*.



- Para crear un programa, empregamos linguaxes de programación.
- Existen múltiples linguaxes de programación na actualidade.
- As linguaxes clasifícanse según:
 - Nivel.
 - De alto nivel.
 - De baixo nivel.
 - Tipo execución.
 - Compilados.
 - Interpretados.



- Para programar, é común usar un entorno de *desenvolvemento integrado*, comunmente denominado **IDE**.
- Os **IDE** son programas que facilitan o proceso de desenvolvemento de aplicacións, e inclúen interfaces gráficas.
- Existen diferentes **IDEs**, como por exemplo, Eclipse, NetBeans, Visual Studio Code...



- As variables son os datos cos que traballan os nosos programas.
- Estas variables almacenan a información na memoria, e permiten o acceso a ditos datos.
- As variables decláranse dando un nome a un tipo de dato

`horas = 2.5`

`mes = 12`

`nome = "Imatia"`

- As linguaxes de programación poden ser tipadas ou non tipadas.
- Java é unha linguaxe tipada.
- Hai que indicar o tipo de dato o que pertence a variable.
- Os tipos poden ser primitivos ou de tipo obxecto.

```
long horas = 2.5
```

```
int mes = 12
```

```
String nome = "Imatia"
```

- Diferente das variables, as constantes permiten almacenar un valor invariable e fixo durante a execución dun programa.
- A declaración da constante, a maiores do tipo de dato, é necesaria por ser unha linguaxe tipado, inclúe a palabra reservada *final*.

Variable	Constante
<code>int primeiroDia = 1</code>	<code>final int PRIMEIRO_DIA = 1</code>

Tipos de datos

Tipos
primitivos

*byte, short,
int, long,
float, double,
char, boolean*

Tipos obxeto

Tipos da
biblioteca
estándar de
Java

*String, Scanner,
TreeSet,
ArrayList...*

Tipos
definidos polo
programador

*Clases propias:
Animal,
Traballador,
Lámpara...*

arrays

*Series de
elementos, coma
vectores ou
matrices*

Tipos
envoltorio

*Byte, Short,
Integer, Long,
Float, Double,
Character,
Boolean*

- Pódense asignar valores as variables e traballar con eles, utilizando o símbolo de igual (=).
- Cambia o valor da variable da esquerda polo resultado da expresión da dereita.

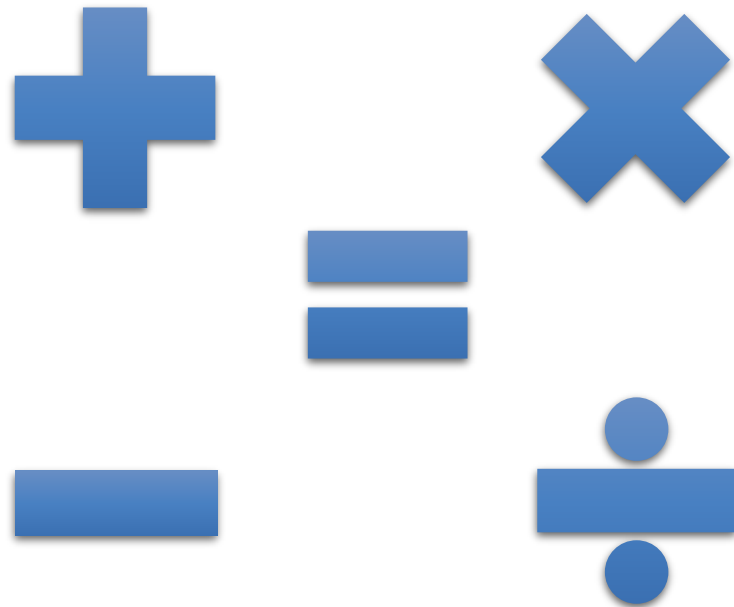
```
int valor = 5;           // 5
valor = valor + 3;       // 8
valor = 2 - valor;       // -6
valor += 3;              // -3
valor -= 1;              // -4
valor *= 4;              // -16
valor = (valor / 2) + 1; // -7
int nuevo = 3;           // 3
int miValor = 2;         // 2
valor = nuevo + miValor;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

- Algúns elementos poden ter múltiples representacións, como cadea, enteiro o double.
- Para pasar dun tipo a outro hai que facer unha conversión (cast).

```
String cnv = "200";  
Integer cnv_int = Integer.parseInt(cnv);  
String cnv_str = Integer.toString(cnv_int);  
Character cnv_chr = cnv.charAt(0);  
String cnv_chr_str = Character.toString(cnv_chr);  
Double cnv_dbl = Double.parseDouble(cnv_str);  
String cnv_dbl_str = String.valueOf(cnv_dbl);  
Float cnvflt = Float.parseFloat(cnv);  
String cnvflt_str = Float.toString(cnvflt);  
Boolean cnv_bool = Boolean.valueOf("true");  
String cnv_bool_str = String.valueOf(cnv_bool);
```

```
// class java.lang.String  
// class java.lang.Integer  
// class java.lang.String  
// class java.lang.Character  
// class java.lang.String  
// class java.lang.Double  
// class java.lang.String  
// class java.lang.Float  
// class java.lang.String  
// class java.lang.Boolean  
// class java.lang.String
```

- Un operador é un símbolo que emprégase para modificar o valor das variables afectadas
- Existen 3 tipos de operadores
 - Operadores aritméticos
 - Operadores lóxicos
 - Operadores a nivel de bit



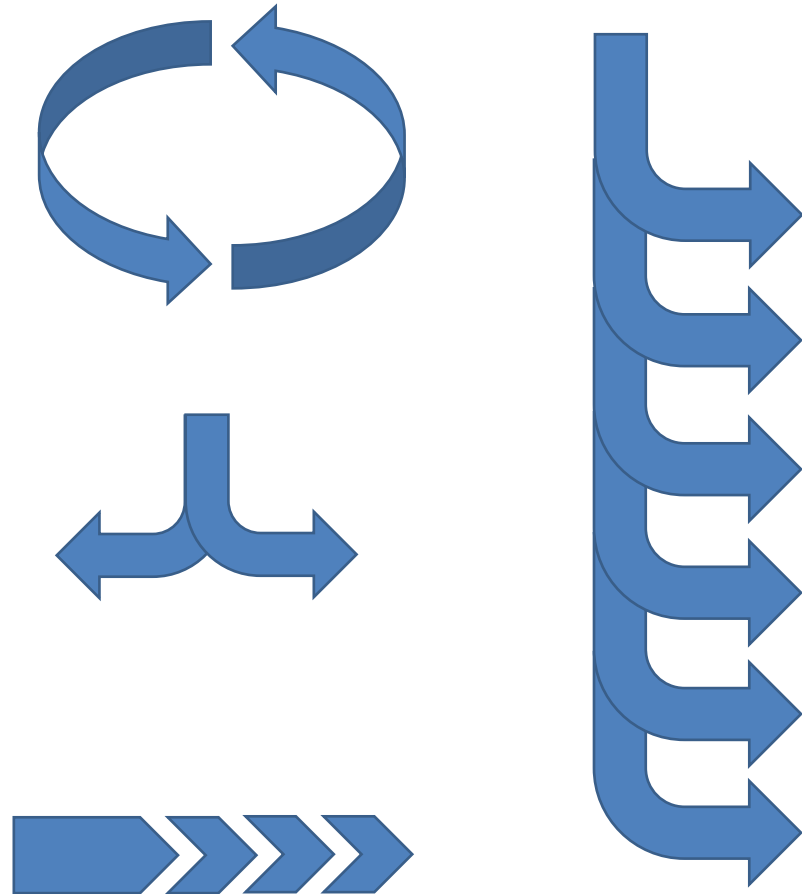
- Os operadores máis comúns son os aritméticos e os operadores lóxicos
- Operadores aritméticos
- Operadores lóxicos

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

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

- Unha función é una serie de instruccións, encapsuladas, que poden recibir unha serie de valores para executar operacións e devolver un valor.
- Un procedemento é unha serie de instruccións, encapsuladas, que poden recibir unha serie de valores para executar operacións que non devolverán ningún valor.
- **Un método é unha serie de instruccións, encapsuladas, que poden recibir unha serie de valores para executar operacións e poden devolver un valor.**

- Las estructuras de control sirven para modificar o fluxo de ejecución dun programa
- Existen estruturas de selección e estruturas iterativas, que poden anidarse entre elas



- As estruturas de control de selección permiten modificar o fluxo dun programa seguindo un determinado esquema
- A estrutura *if* serve para comprobar unha condición, e en base a ela, executar ou non unha 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;
```

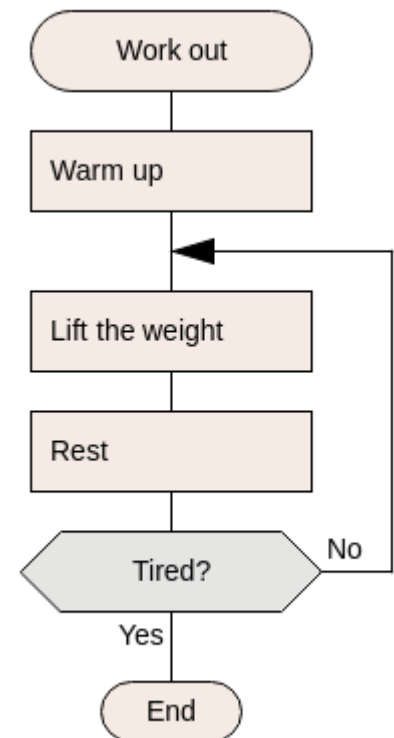
```
if (value >= 5) {
    System.out.println("Maior ou igual a 5");
} else {
    System.out.println("Menor que 4");
}
```


- Outra estrutura, chamada ***switch***, serve para indicar máis de dous 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;  
}
```

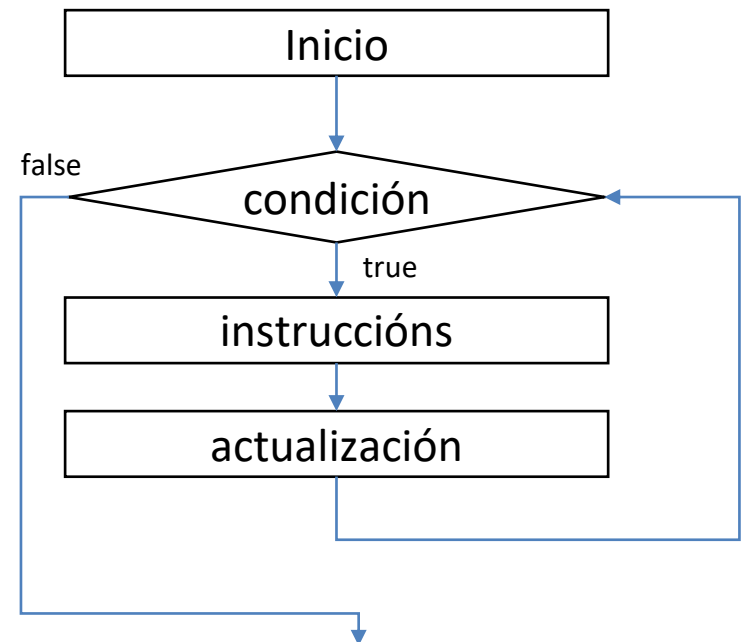
- Denomináanse estruturas iterativas ou **bucles** aquelas estruturas que permitem executar múltiples veces unha sección de código
- Un exemplo de un bucle representado gráficamente



- O bucle **for** utilízase para realizar a execución dun código un número determinado de veces

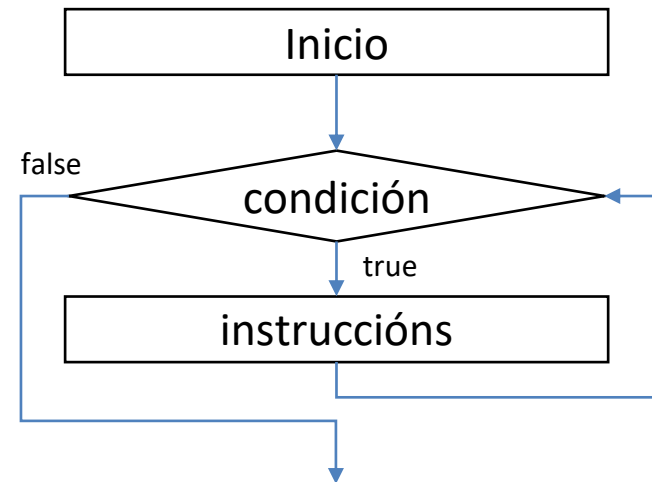
```
for (inicialización; condición; actualización){  
    instrucciones  
}
```

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



- O bucle **while** utilízase para realizar a execución dun código mentras a condición sexa verdadeira

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

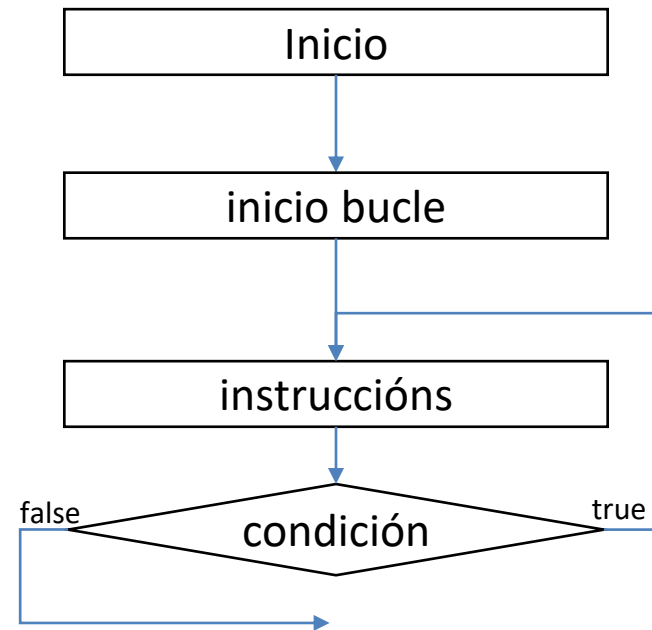


```
int i = 0;
```

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

- O bucle **do-while** utilízase para realizar a execución dun código mentras a condición sexa verdadeira, pero executase como mínimo unha vez

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



- Os bucles poden ocasionar que créese un **bucle infinito**
- A sentencia **break** permite romper a execución do bucle (ou o switch), e o control pase a primeira instrucción fóra da estrutura do control de fluxo
- A sentencia **continue** salta automaticamente a seguinte 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 imprímase a mensaxe de que o valor é 5, o bucle acabará.

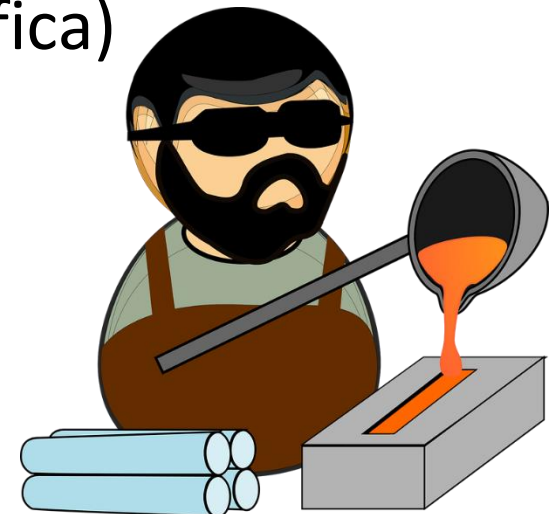
▪ continue

```
int i = 0;

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

- Sólo imprimirase a mensaxe cada vez que o valor de *i* sexa par.

- As clases e os obxectos son os pilares principais da programación orientada a obxectos
- Un símil para entender as clases e os obxectos: As clases son a representación dos elementos do mundo real (clase Persona), e os obxectos son elementos únicos que pertencen a esa representación (unha persona específica)
- Todos os obxectos dunha clase teñen as mesmas cualidades pero con diferente valor (atributos), e pueden facer as mesmas accións (métodos)




```
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++;
    }

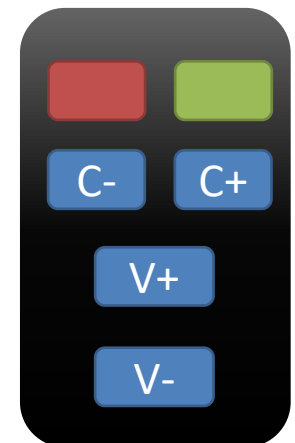
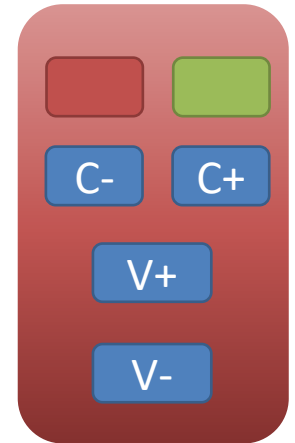
    public void volumeDown() {
        if (this.checkMinValue(this.volume)) {
            this.volume--;
        }
    }

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

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

- A esquerda, un exemplo dunha clase de mando a distancia moi sinxela, onde podemos apreciar os atributos e os métodos
- A dereita e debaixo, a creación dos obxectos utilizando a clase.

```
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());
}
```



- O método co mesmo nome cá clase chamase “constructor” e serve para crear un novo obxecto, una nova instancia da clase.

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

- Este é o método que debemos chamar para crear o obxecto.
- Usando a palabra reservada *new* e o constructor, creamos una nova instancia.
- O constructor pode ter calquer número de parámetros

- Os métodos e atributos dun obxeto teñen una determinada “visibilidade”, isto é, o noso obxeto pode ter uns métodos que non podemos executar (pero pódense executar polo propio método).
- Un exemplo pode ser: Unha cafeteira prepara un café, pero fai unha serie de operacións que teñen que ver co funcionamento interno para preparar o café.
- Da mesma maneira, cando a un obxeto se executa un método, pode chamar a outros método internos, que non teñen que estar dispoñibles de cara o exterior

■ Existen 4 modificadores de acceso

Modificador	Visibilidade
public	Total
protected	A nivel de paquete
<i>Sen modificador</i>	A nivel de paquete
private	Sólo a propia clase

■ Os modificadores pódense aplicar a métodos ou 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) {
        }
    }
```

- Os paquetes en java son contedores que almacenan clases, que agrupan as partes dun programa que, de maneira xeral, tienen unha funcionalidad ou elementos comúns, indicando a ubicación das clases nunha xerarquía de directorios
- Exemplo de paquetes

```
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 {
```

```
package es.imatia.units.resources;  
  
public class Teacher extends Person {
```

```
package es.imatia.units.resources;  
  
public class Doctor extends Person{
```

```
package es.imatia.units.resources;  
  
import java.io.BufferedReader;  
import java.io.InputStreamReader;  
  
public class Input {
```

```
package es.imatia.units.resources;  
  
public class Person {
```

```
package es.imatia.units.resources;  
  
public class PoliceOfficer extends Person{
```

- Exemplo de clase: Crearemos a clase “Coche”, definindo primeiro cales son os atributos que teñen tódolos coches, e cáles son as acción (métodos) que poden facer todos os coches:

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

```
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");  
    }  
}
```

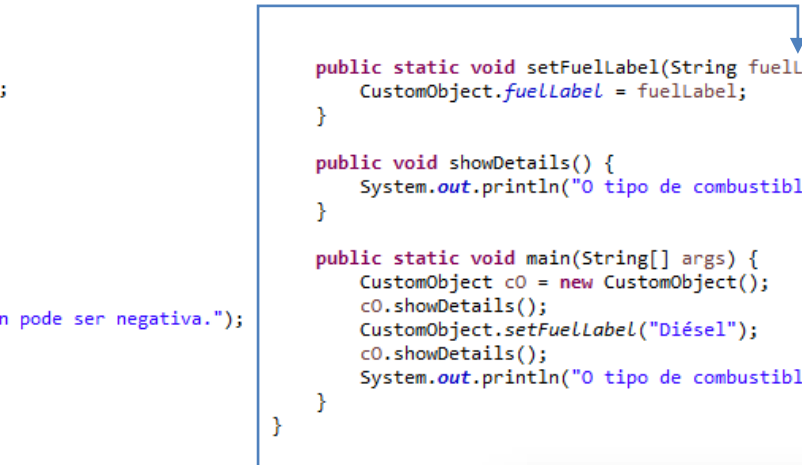
```
public class Car {  
    public String brand;  
    public String model;  
    public static final int MAX_SPEED = 120;  
    public String fuel;  
    public int speedometer = 0;  
    public int tachometer = 0;  
    public String gear = "N";  
    public boolean reverse = false;  
    public int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

- Ademáis dos atributos, créanse tódolos métodos que ten que ter a clase.
- Estes métodos serán os que o obxecto podera facer.
- Un operador moi importante é o operador “this”

- Completamos a clase “Coche”

- Os métodos normalmente invócanse dende un obxeto, pero existen algúns que son invocados dende a propia clase. Da mesma maneira, existen atributos da clase, que son os mesmos para todas as instancias da mesma clase (isto é, tódolos obxetos que pertencen 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 c0 = new CustomObject();  
        c0.showDetails();  
        CustomObject.setFuelLabel("Diésel");  
        c0.showDetails();  
        System.out.println("O tipo de combustible da clase é: " + CustomObject.getFuelLabel());  
    }  
}
```

A blue line with an arrow at the end originates from the 'setFuelLabel' method call in the 'main' method and points to the 'setFuelLabel' method definition. Another blue line with an arrow at the end originates from the 'getFuelLabel' method call in the 'main' method and points to the 'getFuelLabel' method definition. A third blue line with an arrow at the end originates from the 'fuelLabel' attribute access in the 'main' method and points to the 'fuelLabel' attribute declaration in the class.

- Dado o seguinte 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 é a saída 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 estableza una capacidade negativa?

- A encapsulación permite modificar os atributos dunha clase mediante métodos, desta maneira, illamos o acceso os atributos directamente.

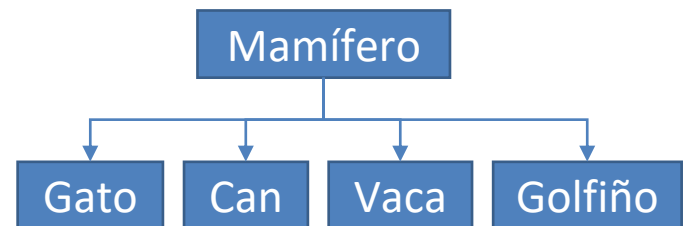
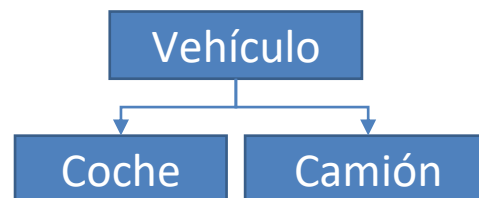
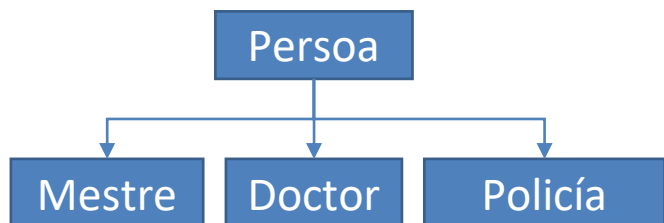
```
public class CustomObject {  
    public int actualFuel = 10;  
  
    public int getActualFuel() {  
        return this.actualFuel;  
    }  
  
    public void setActualFuel(int actualFuel) {  
        if (actualFuel >= 0) {  
            this.actualFuel = actualFuel;  
        } else {  
            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 c0 = new CustomObject();  
        c0.showDetails();  
        System.out.println("Actualización capacidade");  
        c0.setActualFuel(-8);  
        c0.showDetails();  
    }  
}
```

- Estes métodos coñécense como *getters* e *setters*, e a regra común para nomealos é antepoñendo a palabra *set* para establecer valores e *get* para recuperalos (anteponse *is* para recuperar o valor de booleanos)

- O normal cando se usa a encapsulación, e marcar tódolos atributos como *protected* ou *private*, e ter getter e setters públicos para modificalos
- Para obter un bó deseño dun programa, os atributos que sexan públicos teñen que ser constantes inmutables, escribíndose como *public static final*:

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

- A herencia é o sistemas da POO que permite reutilizar clases, añadiendo funcionalidades novas e específicas a clases que xa existen, ou cambiando o comportamento dalgunhas delas.
- A herencia permite que o contido da clase “pai” (superclase) estea contido na clase “filla” (subclase)
- Exemplos de herencia:



- Na orientación a obxectos, as clases Boeing, Harrier, Airbus son subclases de Avión
- As subclases indican una ***especialización*** da clase base, e a súa vez, a superclase é unha ***xeneralización***.
- Para seguir o paradigma da orientación a obxectos, cada subclase ten que ser una especialización da clase base
- Todas as accións referentes as ***xeneralidades*** dunha subclase téñense que executar la clase base

- A extensión da clase indicase coa palabra *extends*.
- Tódalas clases forman parte dunha xerarquía de clases. A clase da que derivan tódalas demais chámase *Object*
- Dado como exemplo as clases Animal, Mamífero, Ave, Golfinho e Falcón, só se indica explicitamente a extensión a partir de Animal, e é nesta onde a extensión de Object é implícita
- As clases heredan os métodos da superclase Object

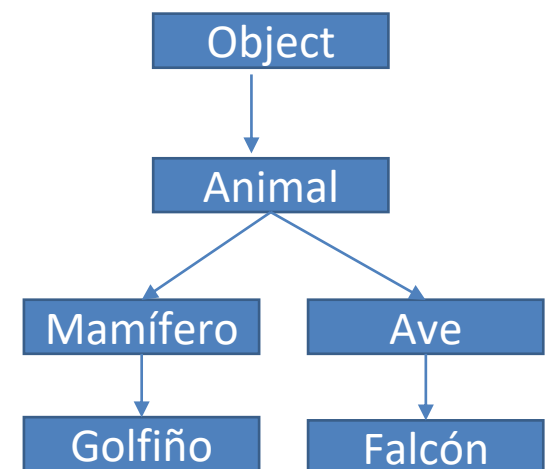
```
public class Animal {
    ...
}

public class Mammal extends Animal{
    ...
}

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

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

public class Falcon extends Bird{
    ...
}
```



■ Exemplo

```
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);
    }
}
```

```
public class PoliceOfficer extends Person{

    protected String squad;

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

```
public class Teacher extends Person {

    protected String area;

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

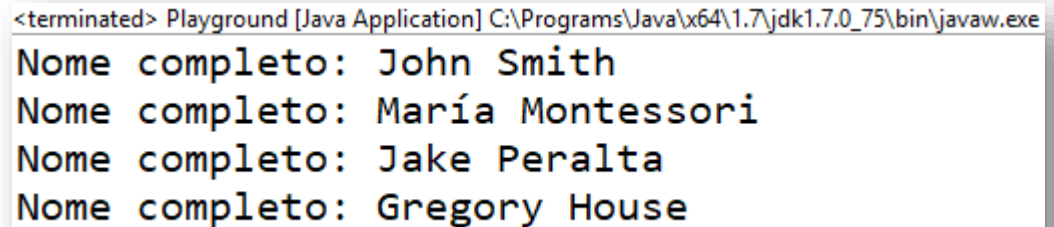
```
public class Doctor extends Person{

    protected String specialization;

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

■ Exemplo:

```
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();  
    }  
}
```

A screenshot of a Java application window titled '<terminated> Playground [Java Application] C:\Programs\Java\x64\1.7\jdk1.7.0_75\bin\javaw.exe'. The window displays the output of the 'getDetails()' method calls for each object: 'Nome completo: John Smith', 'Nome completo: María Montessori', 'Nome completo: Jake Peralta', and 'Nome completo: Gregory House'.

```
<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
```

- Aínda que non se definiu o método *getDetails()* para as clases *Teacher*, *PoliceOfficer* e *Doctor*, este método está definido na clase base, polo que as subclases poden usalo.

- A accesibilidade dos métodos da clase pai ven dada según os seus modificadores.

Modificador	Accesibilidade			
	A mesma clase	Mesmo paquete	Subclase	Outro paquete
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
<i>Sen modificador</i>	Sí	Sí	No	No
private	Sí	No	No	No

- A firma dun método é a combinación do nome do método e o tipo de argumentos recibidos.
- A sobrecarga permite múltiples métodos co mesmo nome pero con diferentes tipos de argumentos
- Tamén existe a 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;  
}
```

- A sobreescritura dun método permite que se modifique a acción da clase base na clase derivada

```
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);
    }
}
```

```
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());
    }
}
```

```
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();
}
```

- É recomendable o uso da anotación *@Override*

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

- As clases abstractas son similares as clases normais, aínda que teñen unha característica especiais, ditas clases abstractas non poden instanciar obxectos.
- A clase abstracta ten a mesma estrutura ca unha clase normal, pero precedida da palabra clave ***abstract*** o inicio da súa declaración
- As clases abstractas poden ter métodos abstractos, que son métodos que non téñ implementación na clase abstracta, pero terán implementación nunha clase concreta.
- Se unha clase ten un método abstracto, ten que ser necesariamente unha clase abstracta

■ Ejemplo de clase abstracta

```
public abstract class Merchandise {

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

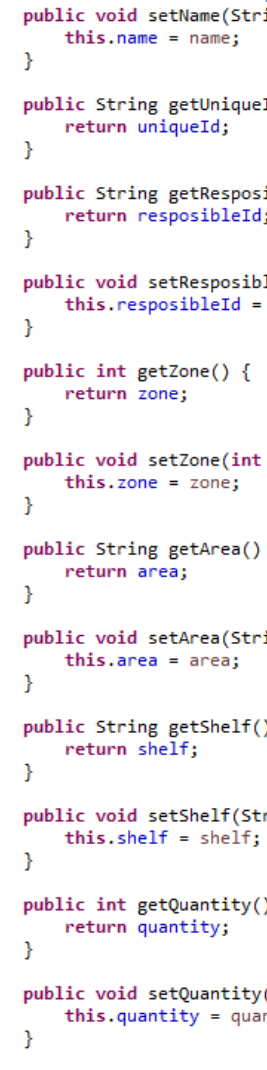
    public Merchandise(String name, String uniqueId, String responsibleId) {
        this.name = name;
        this.uniqueId = uniqueId;
        this.responsibleId = responsibleId;
    }

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

public void setResponsibleId(String responsibleId) {
    this.responsibleId = responsibleId;
}

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;
}
}
```

■ Exemplo de clase concreta que estende dunha abstracta

```
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;
    }

}
```

```
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
```


- A clase final é una clase que non pode ser extendida. Da mesma maneira, un método final non pode ser sobreescrito

```
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;
    }
}
```

- Unha interface non é máis cá unha clase que indica qué accións é obligatorio que poida executar un determinado obxecto dunha clase.
- Os métodos indican unha acción e unha implementación concreta da acción, a interface só mostra a acción.
- Por exemplo, a interface máquina:

```
public interface IMachine {  
    public void start();  
    public void stop();  
    public void maintenance();  
}
```

- Todas as clases que implementen a interface *IMachine*, teñen que ter métodos que implementen as accións de acender, parar e mantemento.

■ Exemplos

```
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");
    }
}
```

```
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 e uso das clases abstractas e as interfaces
- As clases abstractas serven para utilizar un esqueleto de funcionalidades base da nosa clase, pero que haberá certas partes, que por deseño, no se poidan equiparar. Desta forma, só se poderan instanciarse obxectos dunha clase derivada.
- As interfaces mostran o conxunto de accións que poden facer eses obxectos instanciados

- O polimorfismo é a capacidade, dentro dunha relación de herencia, de que calqueira obxecto da superclase pode almacenar un obxecto de calqueira das súas subclases
- A clase padre pode almacenar as clases derivadas, pero non pode darse o 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();  
  
}
```

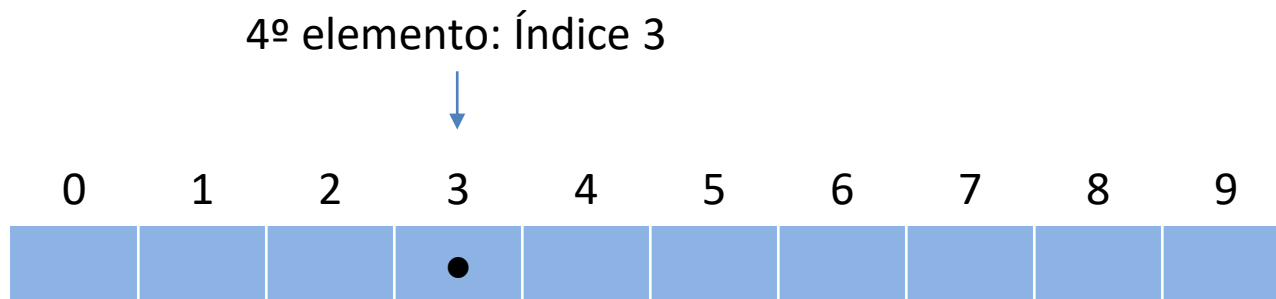
```
Nome completo: John Smith  
Nome completo: María Montessori  
Nome completo: Jake Peralta  
Doctor Gregory House, especialista en nefroloxía e infectoloxía
```

- Da mesma forma, pódese emplear o polimorfismo coas 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
```

- Unha estrutura de almacenamiento serve para almacenar múltiples datos do mesmo tipo.
- Un array é un conxunto de datos almacenados de forma secuencial en memoria
- Accédese os datos mediante un índice, empezando este no índice 0

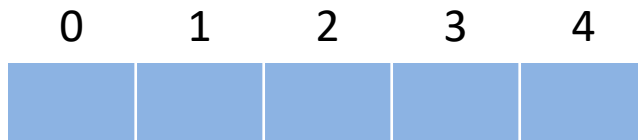


Tamaño del array : 10

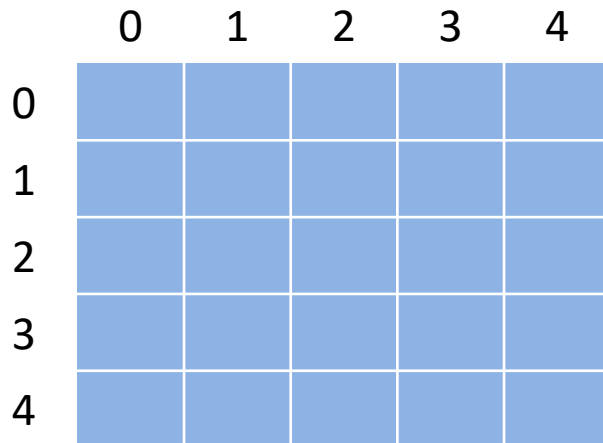
- Na declaração dun array estabécese o tamaño, ou dedúcese según a inicialización empregada.

```
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);  
}
```

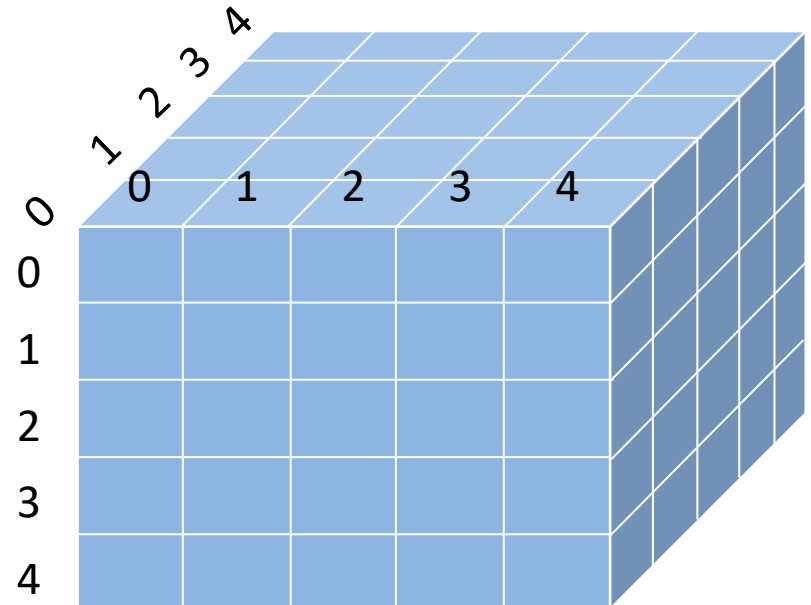

- Os arrays poden ser arrays multidimensionales



```
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, puede usarse 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, pode usarse un bucle for:

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

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

```
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();
}
```

```
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
```

```
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();
}
```

```
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
```

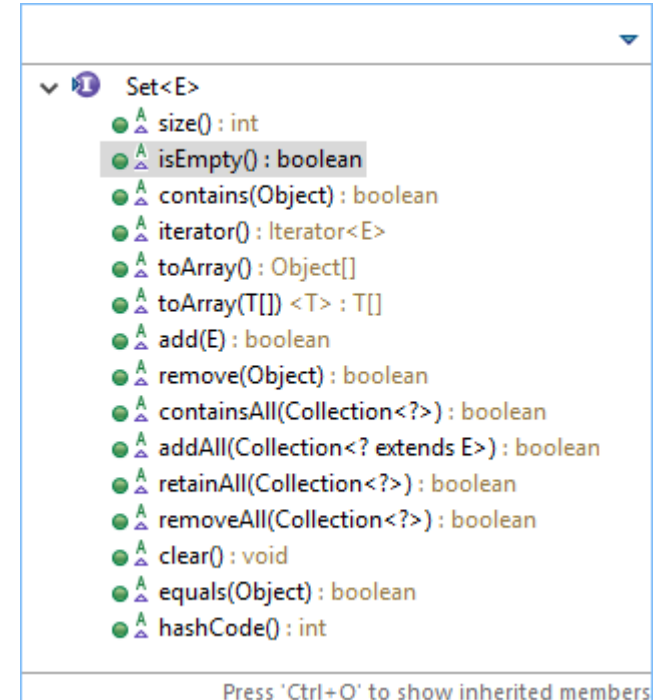
- As coleccións serven para almacenar datos (son estruturas de almacenamiento).
- As coleccións extenden a interfaz *Collection<E>*, e segundo a implementación, permiten almacenar e recorrer a estrutura de diferentes maneiras.

```
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();  
    }  
  
}
```

- As coleccións máis comúns son:
 - Conxuntos
 - Listas
 - Mapas
 - Colas
- Os conxuntos non admiten dous elementos iguais
- As listas permiten múltiples elementos repetidos, e repeta
- Os mapas son coleccións que asocian una clave a un valor. A clave non pode ser asociada a múltiples valores
- As colas só poden manexas os obxetos do principio ou final, dependendo a implementación

- Conxuntos
- Estes son os métodos que teñen os conxuntos
- Os conxuntos máis comúns:
 - 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
- Estes son os métodos que teñen as listas
- Os listas máis comúns:
 - 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
```



The screenshot shows the Java List<E> interface in an IDE. The methods listed are:

- size() : int
- isEmpty() : boolean
- contains(Object) : boolean
- iterator() : Iterator<E>
- toArray() : Object[]
- toArray(T[] <T> : T[])
- add(E) : boolean
- remove(Object) : boolean
- containsAll(Collection<?>) : boolean
- addAll(Collection<? extends E>) : boolean
- addAll(int, Collection<? extends E>) : boolean
- removeAll(Collection<?>) : boolean
- retainAll(Collection<?>) : boolean
- replaceAll(UnaryOperator<E>) : void
- sort(Comparator<? super E>) : void
- clear() : void
- equals(Object) : boolean
- hashCode() : int
- get(int) : E
- set(int, E) : E
- add(int, E) : void
- remove(int) : E
- indexOf(Object) : int
- lastIndexOf(Object) : int
- listIterator() : ListIterator<E>
- listIterator(int) : ListIterator<E>
- subList(int, int) : List<E>
- splitIterator() : SplitIterator<E>

Press 'Ctrl+O' to show inherited members

- Mapas
- Estes son os métodos que teñen os mapas
- Os mapas máis comúns:
 - HashMap
 - LinkedHashMap

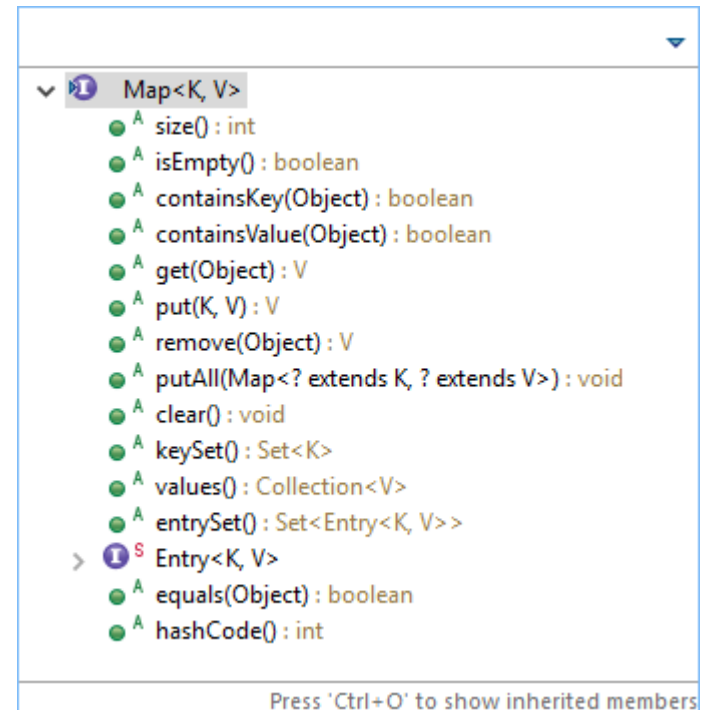
```
public static void main(String[] args) {

    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");

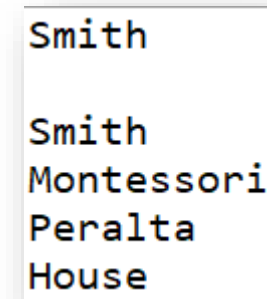
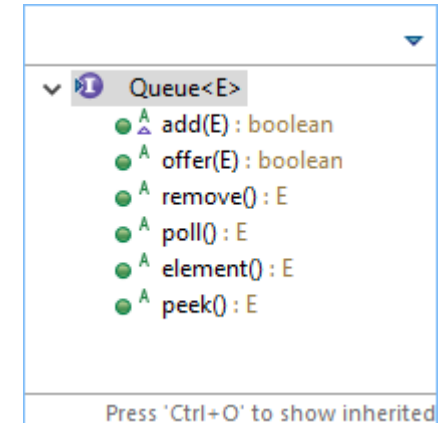
    System.out.println("Contén a clave \"police\": " + customMap.containsKey("police"));
    System.out.println("Contén a clave \"teacher\": " + customMap.containsKey("teacher"));
}
```



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

- Colas
- As colas son listas nas que os elementos introducense e eliminanse por diferentes extremos
- FIFO -> First In - First Out

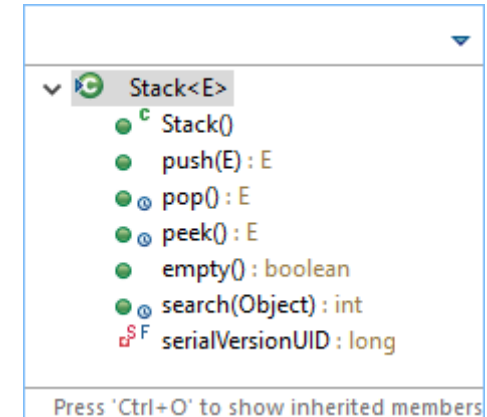
```
public static void main(String[] args) {  
  
    Queue<String> customQueue= new LinkedList<>();  
  
    customQueue.offer("Smith");  
    customQueue.offer("Montessori");  
    customQueue.offer("Peralta");  
    customQueue.offer("House");  
  
    System.out.println(customQueue.peek()+"\n");  
  
    while (!customQueue.isEmpty()) {  
        System.out.println(customQueue.poll());  
    }  
}
```



Smith

Smith
Montessori
Peralta
House

- Pilas
- As pilas son unha colección de elementos que introducense e eliminanse polo mesmo extremo
- LIFO -> Last In – First Out



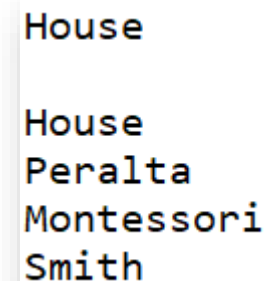
```
public static void main(String[] args) {

    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());
    }
}
```



The diagram illustrates a stack structure. It shows a vertical list of elements: "House", "Peralta", "Montessori", and "Smith". The elements are stacked such that "House" is at the top and "Smith" is at the bottom, demonstrating the Last In – First Out (LIFO) principle.

- A linguaxe XML é unha linguaxe de mercado extensible (*eXtensible Markup Language*).
- A linguaxe XML é unha linguaxe de mercado extensible (*eXtensible Markup Language*).
- É lexible tanto por humanos coma por máquinas
- Independiente da plataforma
- Serve para intercambiar información entre sistemas

```
<?xml version="1.0" encoding="UTF-8" ?>
<component>
  <components>
    <component quantity="1">Tarxeta 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 é un formato lixeiro de intercambio de datos
- **JavaScript Object Notation**
- É simple de leer polas personas e de interpretarse e xearse por máquinas
- Formado por pares conxunto-valor
- Pode utilizarse independente da linguaxe Javascript

```
{
  "component": {
    "components": {
      "component": [
        {
          "-quantity": "1",
          "#text": "Tarxeta 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"
        }
      ]
    }
  }
}
```

- Nun programa de Java pode existir algún tipo de problema (erro) durante a execución do mesmo.
- Cando iso sucede, lánzase una excepción. Pode ser por dividir entre 0, disco duro cheo, intentar acceder a una posición dun vector que non existe, facer 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)
```

```
}
```

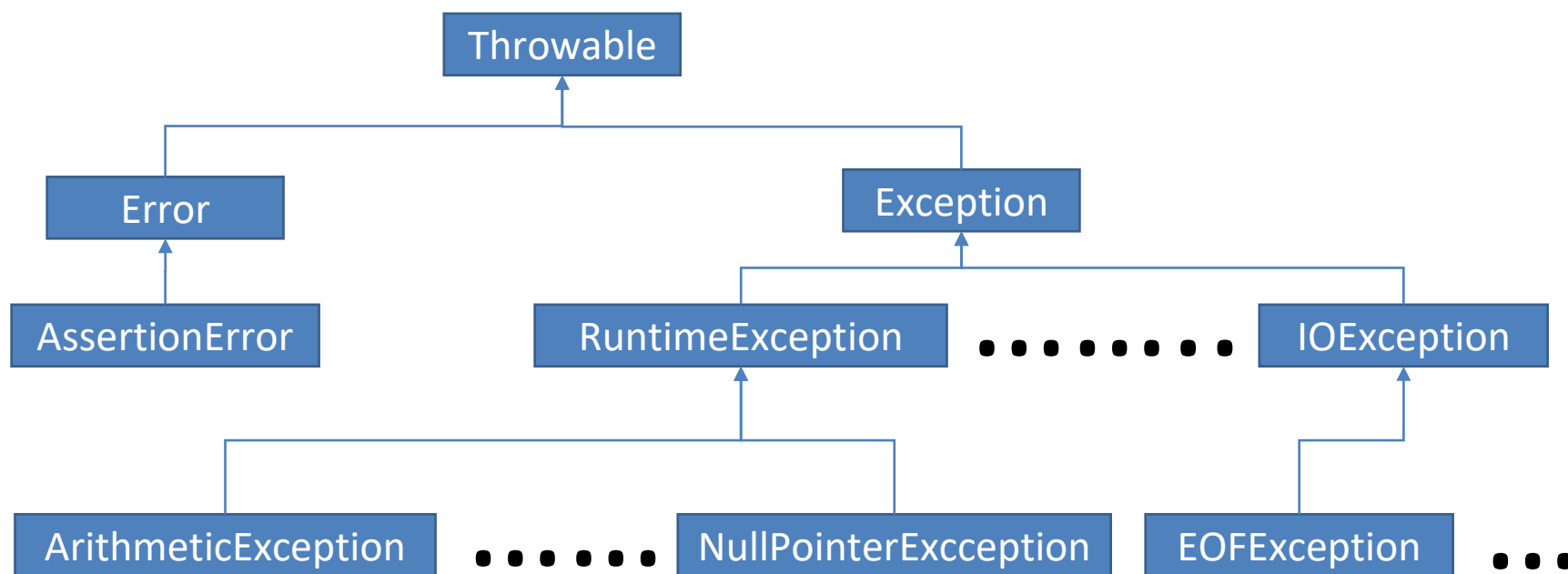
```
public static void main(String[] args) {
```

```
    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)
```

- Cando unha excepción é lanzada, a execución do programa non continúa
- A xerarquía das excepcións é a seguinte:



- Para controlar unha excepción e continuar a execución:
 - 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 controlar unha excepción e continuar a execución:
 - 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)

- As excepción pódense extender para crear excepción propias e lanzar as propias

```
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.");  
    }  
}
```

```
public class ByZeroException extends ArithmeticException{  
  
    public ByZeroException() {  
        super();  
    }  
  
    public ByZeroException(String s) {  
        super(s);  
    }  
}
```

- Nun programa é moi común que se necesite almacenar a información xerada.
- Java permite esa acción mediante o uso de *streams* ou *fluxos*, unha abstracción de todo o que produza ou consuma información.
- Java provee de dous tipos de *streams*:
 - Fluxo de bytes: Orientados o manexo de lectura ou escritura de datos binarios
 - Fluxo de caracteres: Orientados o manexo de lectura ou escritura de caracteres. Internamente, transformase nun fluxo de bytes

- A nosa clase Input que utilizamos recolle información do teclado (*System.in*)
- Para escribir na consola, utilizamos o 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;  
}
```

▪ A lectura e escritura de ficheiros

```
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();  
        }  
    }  
}
```

▪ A lectura e escritura de ficheiros

```
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();  
        }  
    }  
}
```

■ A lectura e escritura de ficheiros

```
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 os ficheiros binarios farase da mesma maneira, pero, no canto de usar os *Reader* ou os *Writer*, úsase os *InputStream* e o *OutputStream*.
- En lugar dos métodos *readLine()* e *println()*, usaránse os métodos *read()* e *write()*
- **FileInputStream, FileOutputStream, FileReader o FileWriter**, fan múltiples chamadas a disco, polo que é mellor utilizar os **BufferedReader, BufferedInputStream, BufferedWriter y BufferedOutputStream**, de maneira intermedia, para minimizar eses 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("Borrarse correctamente");  
    } else {  
        System.out.println("No se eliminó el archivo");  
    }  
}
```