

# JAVA FUNDAMENTALS COURSE

## INHERITANCE IN JAVA



By the expert: Ubaldo Acosta



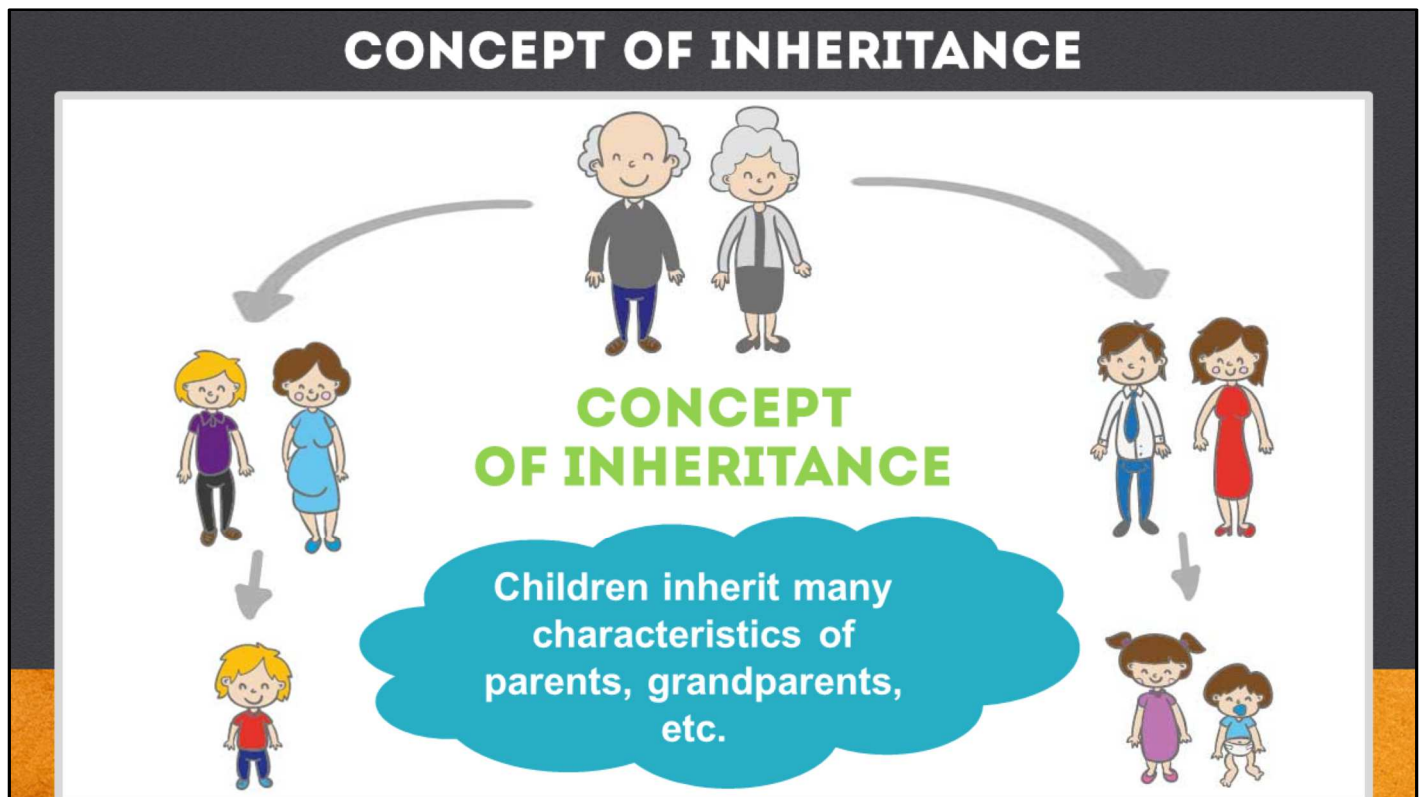
**JAVA FUNDAMENTALS COURSE**

[www.globalmentoring.com.mx](http://www.globalmentoring.com.mx)

Hello, Ubaldo Acosta greets you again. I hope you're ready to start with this lesson.

We are going to study the subject of inheritance in Java.

Are you ready? Come on!



In this lesson we are going to study the subject of inheritance.

One of the ways in which we understand inheritance, are the characteristics that are maintained from generation to generation in a family. For example the grandparents had blond hair, and one of the children inherits this characteristic, in turn the son has a son who also inherits the blonde hair.

On the other hand the same grandparents could have had a daughter with brown and straight hair, and the grandchildren can also inherit this same characteristic of having brown and straight hair.

In object-oriented programming, the concept of inheritance is exactly the same. We will define a hierarchy of classes that will allow us to inherit characteristics between Parent classes and Children classes.

Let's see some particular features of Java below.

# INHERITANCE IN JAVA

**REPRESENT  
COMMON  
BEHAVIOR**

**IT IS A  
CHARACTERISTIC  
OF THE POO**

**AVOID CODE  
DUPLICATION**

**DESIGN  
CLASSES MORE  
SIMILAR TO THE  
REAL WORLD**

**JAVA FUNDAMENTALS COURSE**  
[www.globalmentoring.com.mx](http://www.globalmentoring.com.mx)

Inheritance is one of the most important concepts in object-oriented programming (OOP), and we will mention some of the most important reasons why we will apply this concept in the design of our classes.

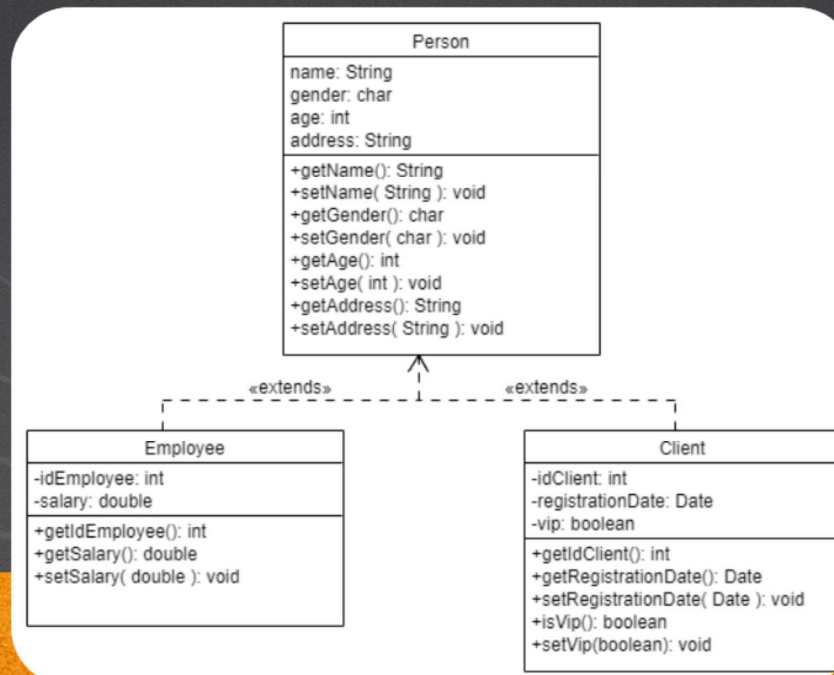
The inheritance will allow us to represent characteristics or behavior in common between classes, allowing us to define in the Parent class the attributes or methods that are common to the children classes, which will inherit these attributes or methods defined in the Parent class.

This allows us to avoid duplicating the code between the Father class and the Children classes, so we comply with the reuse of code which is one of the main objectives of the OOP.

If we apply the concept of inheritance, we can model real world systems in a simpler way. If the definition of objects is already a great contribution, designing a hierarchy of classes will make it even easier for us to model classes.



## EJEMPLO DE HERENCIA EN JAVA



Let's see an example of inheritance in Java. We can observe a class called **Person**, which like any person has many characteristics, but we will focus on a few. A person has a name, a gender, an age and an address, we will capture these characteristics in each of the corresponding attributes in the Java class, you can have many more, but we will focus on these to simplify the example.

If we are trying to model a sales system, we could have some variants of a person in our system, for example the **Employee** and **Client** class.

On one hand, unlike a simple **Person**, an **Employee** has a salary, and his activity can define the type of employee he is. Salesman, Manager, however we will not go into more detail in this example, we will leave the inheritance example only up to the **Person - Employee** relationship. An employee also has an identifier, which is unique to each employee, and the **Employee** class has the methods for each attribute.

On the other hand, we can have a **Client** that registers in the Sales system, this **Client** can have as attributes an identifier generated in a very particular way for him, we can also have a date of registration, and we can also indicate if it is a VIP (Very Important Person), for such cases, we assign the variables described in the **Client** class, as well as the respective methods.

In this example we are not indicating how we arrived at the design of these classes, however the general process is to identify the actors or objects (entities that will generally be our classes) that will interact in our system, and from the needs of our system we can go scoring which are the characteristics (attributes) and functionality (methods) that each registered entity must have.

Once these entities have been found with their attributes and methods, we see if there are characteristics or functionality in common, and that is precisely how we come to this type of design. We can see that the **Person** class is the Parent class and that the **Employee** and **Client** classes extend from the **Person** class, that is, that they inherit the characteristics of the **Person** class.

This allows that we do not have to repeat the code already defined in the **Person** class, and once we create an object of type **Employee** or **Client**, in automatic it will also have the characteristics of the **Person** class. In the next slide we will see a more visual example of this.

## EXAMPLE OF INHERITANCE IN JAVA

### Full Employee Class

#### Person Class

+ getName()  
+ setName()  
+ getGender(), etc

#### Employee Class

+getIdEmployee()  
+getSalary(),  
+setSalary(), etc

\_emp1: Person

idEmployee: int="36548"  
salary: double="2500.00"

The created object of type Employee contains the methods of the Person class which are heritable:

+ getName ()  
+ setName ()  
+ getGenero (), etc

It also has its own methods:

+ getIdEmployee ()  
+ getSalary ()  
+ set Salary ()

In this picture we can see on the left side the definition of two classes with inheritance. This is a way in which we can visually imagine what happens with the definition of classes once we handle the concept of inheritance.

What we visualize is that the Person class has private attributes and several public methods. Private attributes are not inherited, as are private methods. That is, while we use the private access modifier, the attribute or method marked with this modifier will not be inherited.

But the public modifier is inherited. Later we will mention what happens with the modifiers of type package and protected, at the moment we will handle only these two modifiers, private and public to simplify the explanation of the concept of inheritance.

As we observe the figure shows the "complete" definition of the Employee class, which when inheriting from the Person class, inherits the characteristics that are not private, that is, it does not inherit the attributes since they were defined as private, but it inherits the methods defined as public. This allows that when we create an object of employee type, such as the one described on the right side of the figure, we can see that this object of type Employee has access both to the methods inherited from the Person class, as well as to the methods defined in it. Employee class.

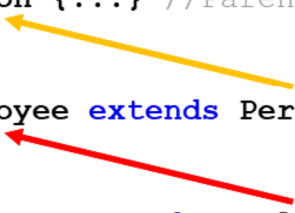
We should note that it can be inherited either attributes or a methods, what restricts it is the access modifier, either private or public. Remember that there are 4 access modifiers in total, but for now we will simplify with just these two.

In summary, when we visualize a class that inherits from another class, we can imagine this class as the set of its own attributes and methods, as well as the sum of the inheritable attributes or methods of the Parent class or classes, since it can receive inheritance from a Parent class, or a "GrandParent" class, that is, a Parent class of his own Parent class. We can have as many levels of inheritance as needed.

# INHERITANCE SYNTAX IN JAVA

## Inheritance Syntax in Java:

```
class Person {...} //Parent Class definition (super class)
class Employee extends Person {...} //Child class definition (subclass)
class Manager extends Employee {...} //Another child class definition (subclass)
```



**Note: ALL classes that do not explicitly specify a extends, then they inherit from the Object class, which is the Java core class.**

**JAVA FUNDAMENTALS COURSE**  
www.globalmentoring.com.mx

To define the inheritance in Java, simply use the word **extends** in the definition of the Java class and indicate the name of the parent class. Eg.

```
class Employee extends Person
```

It should be noted that in Java the inheritance is simple, this means that **we can only inherit from one class at a time**. There may be a hierarchy of classes upwards, for example Class GrandParent, Parent and Child, but a class can not inherit from a class Father and a class Mother, that is, from two classes at the same time, only by class hierarchy .

This IS allowed, define a Class Hierarchy (Parent Class or SuperClass, Child Class or SubClass):

```
class Person
```

```
class Employee extends Person
```

```
class Manager extends Employee
```

This is NOT allowed (you can not inherit two classes at once, in Java the inheritance is simple):

```
class Manager extends Employee, Person
```

It is important to indicate that ALL classes in Java inherit from the Object class, indirectly or directly. In the case of the Person class, since it is not indicating that it inherits from any class, that is, it does not use extends in the definition of its class, it is understood that it automatically inherits from the Object class. But the classes Employee or Manager, because they extend from other classes, will inherit the methods of the Object class indirectly, but all Java classes in one way or another inherit the characteristics of the Object class. Finally, the Object class is the core class of Java, and belongs to the java.lang package. Later we will talk about the theme of packages in Java.

## USE OF SUPER AND THIS

```
class Person { //Parent class definition (superClass)

    //One argument constructor
    public Person(String name){
        this.name = name;
    }
}

class Employee extends Person { //children class (subClass)

    public Employee(String name, double salary) {
        super(name); //Super must be the first line of the constructor
        this.salary = salary;
    }

    //Test creating and employee object
    public static void main(String[] args) {
        Employee e = new Employee("John", 2500);
    }
}
```

If we apply the concept of inheritance, the Child class or Subclass can access the allowed attributes, methods or constructors of the parent class just by using the **super** keyword.

In this way we can take legacy code and take advantage of it, for example, to start an object of the child class. This is very common to use it inside the constructor to call the constructor of the parent class from the constructor of the child class.

The constructor of the child class will only do the work corresponding to the child class, such as the assignment of attributes, but it will not do the work that the constructor of the Parent class already performs if it is possible to use it. For that, the super keyword is used.

In fact the **this** keyword can be used identically but to call a constructor of the same class, and the **super** keyword to call the constructors of the Parent class. **super** not only lets you call constructors of the parent class, but any attribute, method or inheritable constructor of the parent class.

In the example we can observe that we use the word super to send to call the constructor of the Parent class from the Child class.



## toString AND super

```
//Extract code from Person and Employee classes
public class Person {
    @Override
    public String toString() {
        return "Person{" + "idPerson=" + idPerson
            + ", name=" + name
            + ", age=" + age + '}';
    }
}

public class Employee extends Person {
    @Override
    public String toString() {
        //First we call the toString method of the Person class
        //to see the parent class values,
        //after we print the child attributes of the child class
        return super.toString() + " Employee{salary=" + salary + "}";
    }
}
```

In the code, we can see an example for the use of the `toString ()` method. The method `toString ()` as we have said, we will use to show the state of an object, that is, the values of the attributes at a certain time of the object's lifetime.

The `toString` method is a method inherited from the object class. The Object class is the Parent class of all Java classes, either explicitly or implicitly, as we have commented. For example, in the case of the Employee class, the Object class would be the grandparent class of the Employee class, because the Person class has been explicitly defined as its Parent class, and because the Parent class has not indicated that it extends of some kind, then it is understood that its parent class is the Object class. In this way, the Employee class also receives the methods and attributes that it inherits from the Object class.

To indicate that a method belongs to a parent or higher level class, such as the Object class, then we must add the annotation `@Override`, this means that we are overwriting the behavior of a method of the parent class or higher classes. However, this topic of overwriting will be studied in the next level. But for the example of this course we will only say that it is necessary to add this annotation so that we indicate to the Java compiler that we know that it is a method defined in a superior class, in this case it is a public method of the Object class.

Now, this method `toString ()` serves us as we have said, to convert the state of an object to a string. But for the case we are showing, the Person class has its `toString ()` method and ideally we would reuse this code to complete the `toString` method of the Employee class, since if we combine the `toString` method of the Person class with the class of the Employee class, then we can show the status of all the attributes of the Employee class and of its parent class Person.

How do we do this? As we observed in the code, what we do is use the word `super`, and because `toString` is a public method redefined (overwritten) in the Person class, then we can access this method of the parent class by means of the word `super`, the dot operator and the name of the method that we want to access from the parent. This will return a string with the status of the Person object, and with that, it is enough to concatenate the value that we have not yet placed in the Employee class, so we concatenate the value of the salary variable and now if we can return the entire chain to that now the `toString` method of the Employee class can show both its own status and the status of each attribute of its parent class that is Person.

We will put this into practice in the following exercise.



**ONLINE COURSE**

# **JAVA FUNDAMENTALS**

Author: Ubaldo Acosta



**JAVA FUNDAMENTALS COURSE**

[www.globalmentoring.com.mx](http://www.globalmentoring.com.mx)

