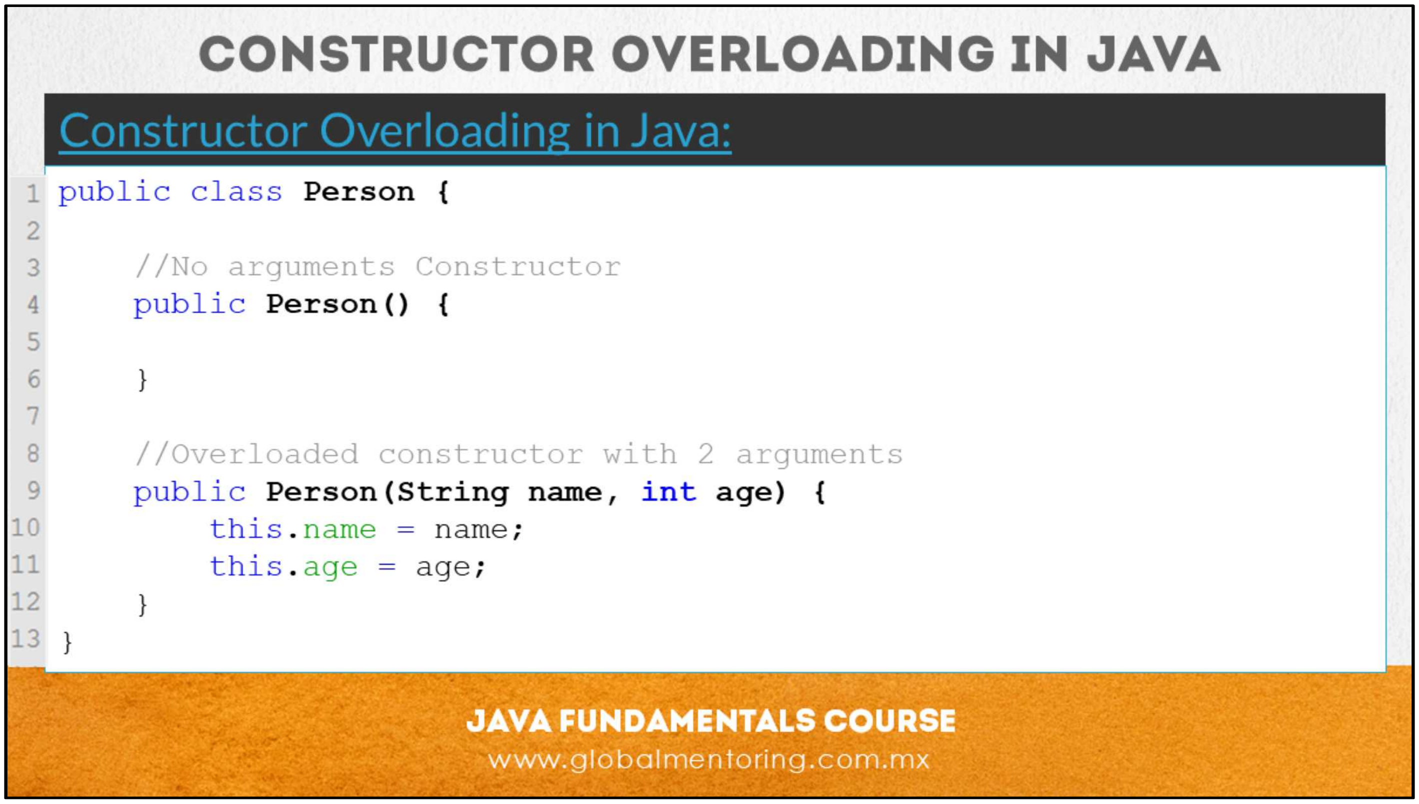
Hello, Ubaldo Acosta greets you. Welcome or welcome again. I hope you're ready to start with this lesson.

We are going to study the topic of Constructor Overload in Java.

Are you ready? OK let's go!



CONSTRUCTOR OVERLOADING IN JAVA

Constructor Overloading in Java:

```
1  public class Person {
2
3      //No arguments Constructor
4      public Person() {
5
6      }
7
8      //Overloaded constructor with 2 arguments
9      public Person(String name, int age) {
10         this.name = name;
11         this.age = age;
12     }
13 }
```

JAVA FUNDAMENTALS COURSE
www.globalmentoring.com.mx

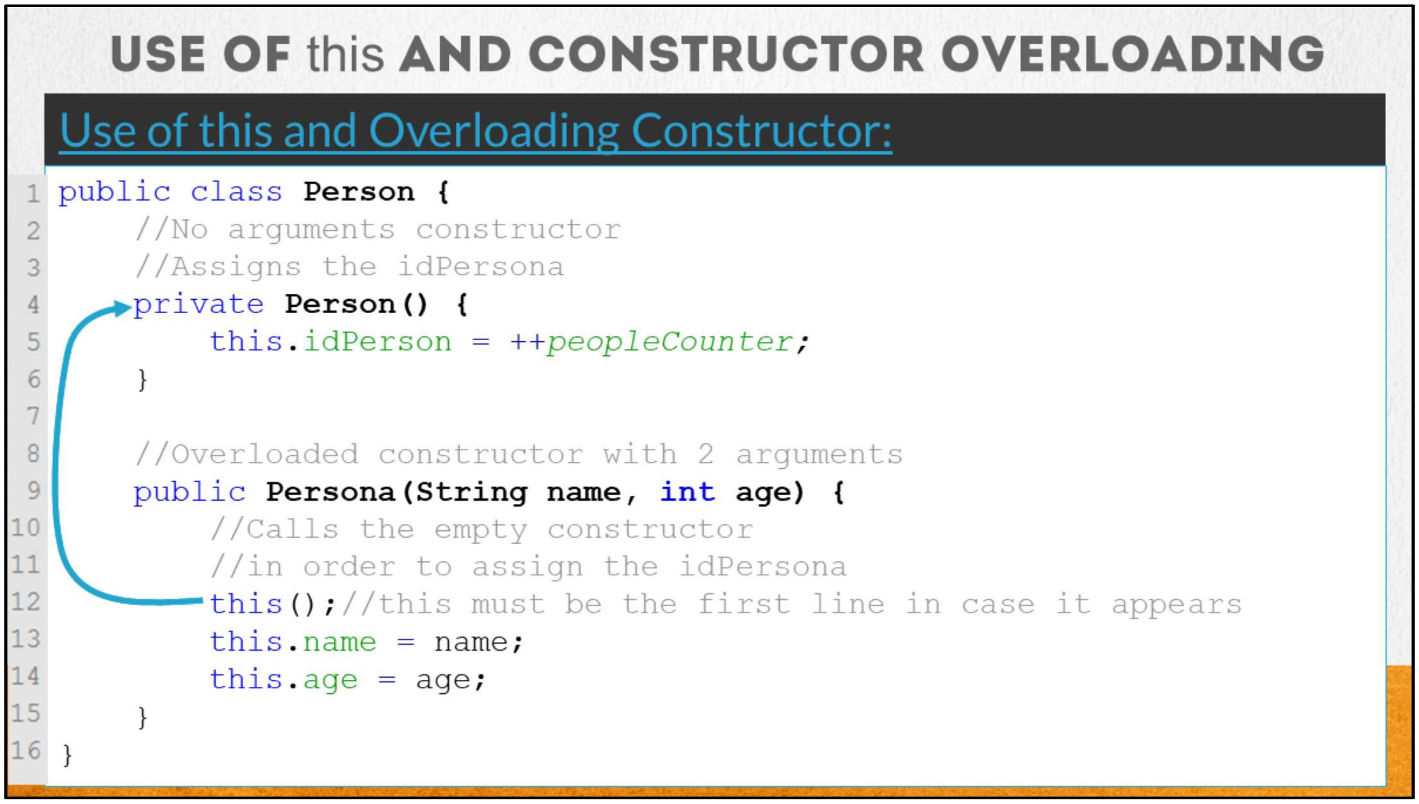
In this lesson we will study the concept of Constructor Overloading in Java

First, the overload of Constructors has to do with offering different options to create an object of a particular class, this will be achieved by creating different constructors with different arguments, either in number or in data type. Recall that the name of the constructor is equal to the name of the class, respecting that Java is case-sensitive. Therefore whenever we find a method that is equal to the name of the class and that does not return any type of data, then it will be a Constructor of the class, and there may be several in the same class.

We can say then, that the overload of a Constructor is to offer more options to be able to construct an object of a class. In the code shown we can see that the empty Person constructor (line 4) was created first, that is, without arguments. However, many times we want to offer more options to build a Person type object and from the moment of creation force to provide the name and age data.

Therefore, we add an overloaded constructor, with 2 arguments (line 9), so we have Person (String, int). The important thing when overloading a Constructor is to observe the types of the arguments, and not their names, so there should be no other constructor with the same types already defined in the same order in which it is defined. However, if we add a Constructor named Person with the inverted arguments, that is, Person (int, String) then this is considered another overload of the already defined constructors. That is to say that the order and type of the arguments does matter.

In summary, the overload of Constructors, is to define a Constructor with the same name of the class, but with different arguments, considering the type and order of the arguments. This is done with the aim of providing several options for the creation of our objects according to the class we are coding.



USE OF this AND CONSTRUCTOR OVERLOADING

Use of this and Overloading Constructor:

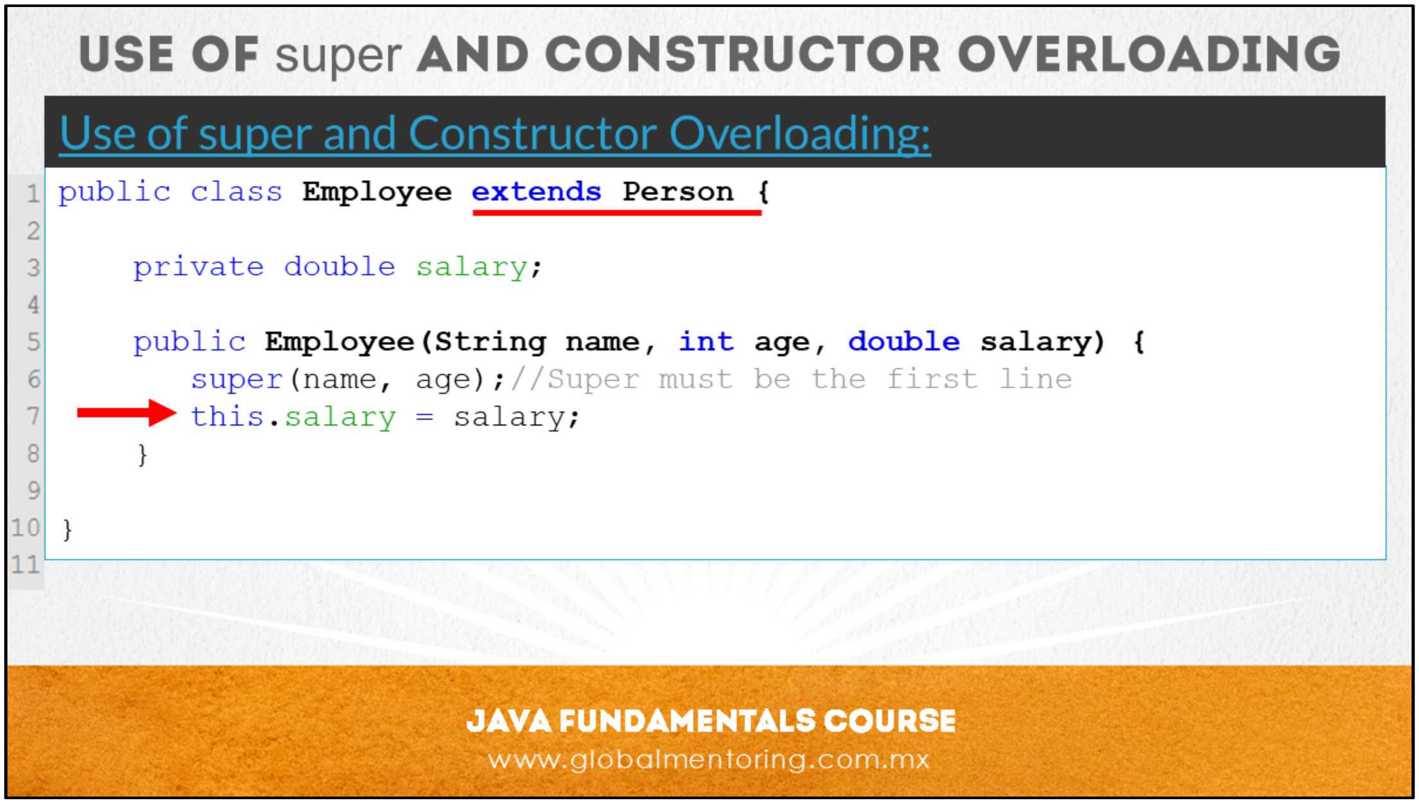
```
1  public class Person {
2      //No arguments constructor
3      //Assigns the idPersona
4      private Person() {
5          this.idPerson = ++peopleCounter;
6      }
7
8      //Overloaded constructor with 2 arguments
9      public Persona(String name, int age) {
10         //Calls the empty constructor
11         //in order to assign the idPersona
12         this();//this must be the first line in case it appears
13         this.name = name;
14         this.age = age;
15     }
16 }
```

Sometimes we can have calls between constructors of the same class. There are several reasons to perform this task, but suppose that the empty constructor does the task of assigning a value to the idPerson attribute, then this code can avoid repeating it in other constructors only by calling the empty constructor from any other overloaded constructor.

To make this call between constructors, we use the **this** operator and later we specify the arguments that we send according to the constructor that we want to use. In our example, we are calling the empty constructor (line 12) in order to reuse the code that the empty constructor already performs. In fact we can see as an extra feature that the empty constructor is private, that is, we can not create objects of type Person without arguments, but we must necessarily use some constructor overloaded with arguments, in this case we must provide the name and age of the person in a mandatory way, and internally the functionality of assigning the value of idPersona is done internally by the Person class through the empty constructor, therefore the empty constructor is called by means of **this( )**, this means that the constructor is called without arguments. If we had a constructor with only one argument of type String and we would like to send it to call from another constructor, the syntax would be: **this ("String");** that is, we should provide a string argument to call this argument.

The only condition for calling another constructor from an overloaded constructor is that using this is the first line of the constructor, so line 12 the first thing used in the constructor is **this( );**

It is also possible to use this characteristic with Children classes using the super operator, that is, when we are managing inheritance, let's see how this task is performed.



USE OF super AND CONSTRUCTOR OVERLOADING

Use of super and Constructor Overloading:

```java
public class Employee extends Person {

    private double salary;

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

}
```

JAVA FUNDAMENTALS COURSE
www.globalmentoring.com.mx

Starting from the Person class that we have in the previous slide, we will define the Employee class, which extends from the Person class. Therefore, it inherits all the characteristics that are inheritable, that is, the ones that are not private.

This means that the Employee class has access to the public constructor Person of two arguments (String and int), and therefore we can rely on it to initialize the attributes of the Person class. In fact, because of the way the Persona class is built, it is the only way to initialize the attributes of that class.

For example, the idPerson attribute of the Person class, there is no way to access it directly, even through a method or a constructor. The only way to access and initialize it is by calling the empty constructor, however this empty constructor of the Person class is private, therefore we can not use it from the Employee class.

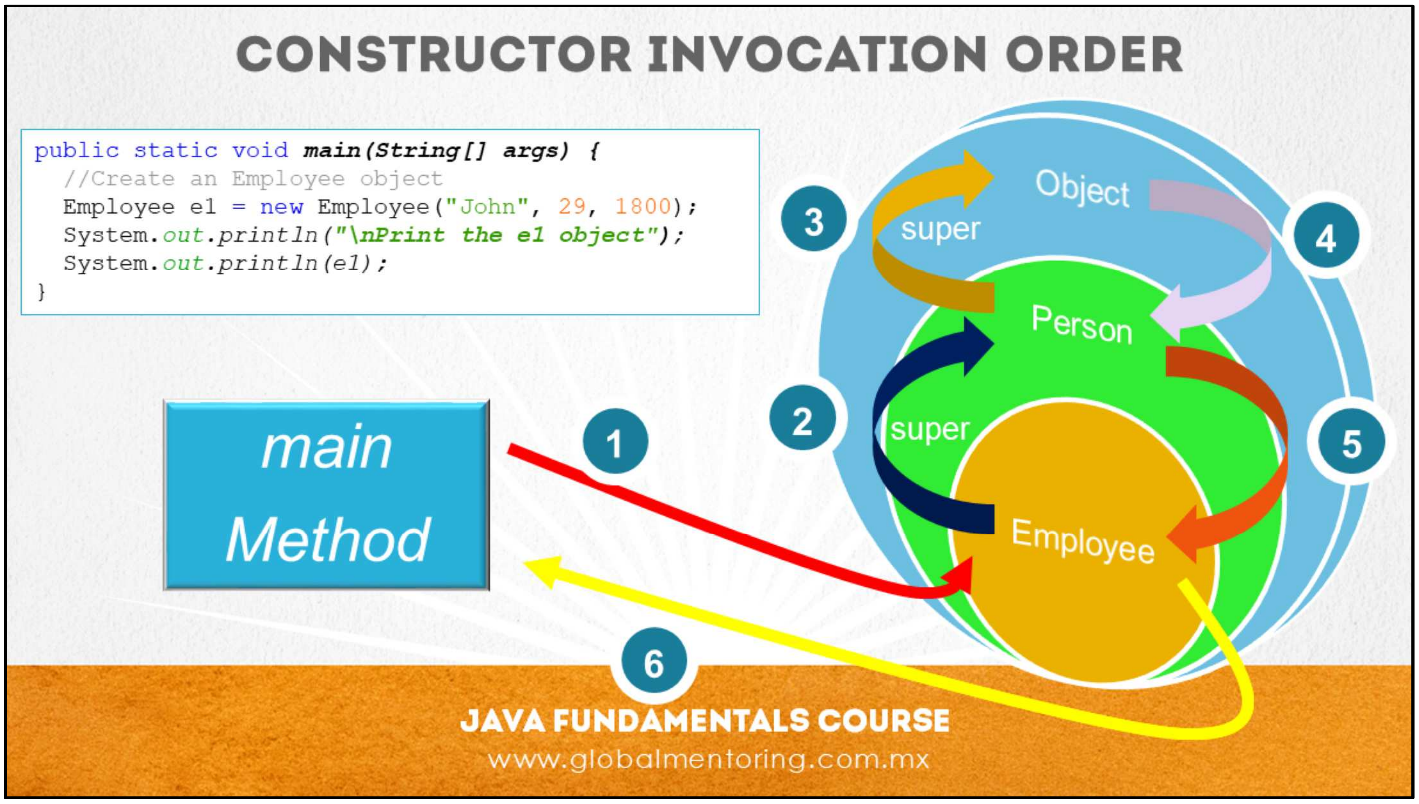
The only way we have to initialize the Person object is through the call to the public constructor Person (String name, int age). Now, the question is how can we access this constructor?

The answer is using the **super** operator. The **super** operator allows us to access the constructors, methods or attributes of the parent class as long as they are accessible to our class. When we see in detail the topic of access modifiers we will see that there are other modifiers besides public that will allow us to access Constructors, methods or attributes of the parent class.

However, for the example we are showing, the **super** operator is being used to initialize the attributes of the Person class, from which Employee extends, if we did not do it in this way, we would be leaving the values assigned to the attributes of the Person class, since all its attributes are private, and values could be assigned if mutator (set) methods exist for the respective attributes, but if this mutator method does not exist then they would be uninitialized.

So this is an example of how we can start to configure our classes so that when creating our objects we start putting restrictions on how they should be used and initialize their attributes.

Finally, we can observe that each class perform its work, that is, the Person class performs the respective tasks when creating an object of type Person, or the assignment of the variables for the children classes, and the children classes, in this case the Employee class does the initialization of its attributes. In this way we avoid code duplication and begin to create more elaborate and functional designs in the creation and coding of our classes.



CONSTRUCTOR INVOCATION ORDER

```
public static void main(String[] args) {
  //Create an Employee object
  Employee e1 = new Employee("John", 29, 1800);
  System.out.println("\nPrint the e1 object");
  System.out.println(e1);
}
```

With what we have seen so far, we can realize not only how our Java classes are configured, but also the order in which Java objects are created.

First, the constructor of the object we are creating is called. When you start running this constructor, an implicit call is made to super (), that is, to the Parent class.

All classes in Java inherit from the Object class as we have commented, therefore all constructors at one time or another make a call to super (), and eventually end up calling the empty constructor of the Object class. This constructor is responsible for reserving memory among several other tasks, however for this lesson it is enough to know that this is the constructor that is called in all cases when an object is created.

Once the constructor of the parent class finishes executing, the control is returned to the constructor that had initiated the call, or which is in turn the parent constructor, and so on until the constructor chain is completed. In a few words the constructors children are the ones who initiate the call, but these in turn in their first line of code, directly or indirectly call the constructor of the parent class. If nothing is specified, then the call is to the empty constructor of the parent class, that is, super ();

Once the empty constructor of the parent class finishes executing, the control is returned to the constructor of the class that started the call and continues with the execution of its code. If the constructors call has already been concluded, then the control is returned to the method that made the call to the construction of the respective object.

For example, in the code of our example, the main method creates an object of the class Employee, but this in turn has as a parent class the Person class, and the person class does not explicitly extend any other class. , then implicitly extends the Object class. All the classes in Java if you do not specify a extends in the definition of your class, then we can say that your parent class is the Object class as we have commented.

Therefore, the first call that is made is to the constructor of the Employee class, with 3 arguments, that is, Employee (String, int, int). This code is going to assume that it is the same as the previous slide. Then, before making the allocation of the salary attribute, the first thing it does is make a call to the constructor of its parent class, that is, to the constructor of the Person class. Because an explicit call is being made by super, then the constructor of the Person class is called with two arguments.

This constructor of the Person class (String, int) will suppose that it is the same code of the previous slide, therefore, a call to the constructor without arguments is made with the use of this (). And it is a constructor without arguments because it does not make an explicit call to super, so implicitly it is called super (), that is, the constructor of the Object class. Once this constructor of the object class finishes executing its code, it returns the control to the empty constructor of the Persona class () and continues a line after where the implicit call was made to super (). And once they finish executing the codes of the constructors of the Person class, it returns the control to the constructor of the Employee class, to finally conclude with the creation of the Employee object, which has already initialized all the attributes of both the Employee class and of the Person class, according to the process described above.

This is important to keep in mind since when we are programming our constructors and considering the initialization of variables it is important to know the order in which our constructors will be executed, as well as the creation of the classes, especially when we are applying the concept of inheritance.

ONLINE COURSE

JAVA FUNDAMENTALS

Author: Ubaldo Acosta

Global Mentoring
JAVA
UNIVERSITY

JAVA FUNDAMENTALS COURSE

www.globalmentoring.com.mx