
Constructors in Java

A Java *constructor* is special method that is called when an object is instantiated. In other words, when you use the `new` keyword. The purpose of a Java constructor is to initializes the newly created object before it is used.
Here is a simple example that creates an object, which results in the class constructor being called:

```
MyClass myClassObj = new MyClass();
```

This example results in a new `MyClass` object being created, and the no-arg constructor of `MyClass` to be called. You will learn what the no-arg constructor is later.
A Java class constructor initializes instances (objects) of that class. Typically, the constructor initializes the fields of the object that need initialization. Java constructors can also take parameters, so fields can be initialized in the object at creation time.

Defining a Constructor in Java

Here is a simple Java constructor declaration example. The example shows a very simple Java class with a single constructor.

```
public class MyClass {
public MyClass() {    }
}
```

The constructor is this part:

```
   public MyClass() {    }
```

The first part of a Java constructor declaration is an access modifier. The access modifier have the same meanings as for methods and fields. They determine what classes can access (call) the constructor.
The second part of a Java constructor declaration is the name of the class the constructor belongs to. Using the class name for the constructor signals to the Java compiler that this is a constructor. Also notice that the constructor has no return type, like other methods have.
The third part of a Java constructor declaration is a list of parameters the constructor can take. The constructor parameters are declared inside the parentheses `()` after the class name part of the constructor . In the constructor declaration example above no parameters are declared. I will show an example of a Java constructor declaration with parameters later in this text.
The fourth part of a Java constructor declaration is the body of the constructor. The body of the constructor is defined inside the curly brackets `{ }` after the parameter list. In the constructor example above the constructor has no operations inside the constructor body. It is said to be an "empty" constructor.

Constructor Overloading - Multiple Constructors for a Java Class

A class can have multiple constructors, as long as their signature (the parameters they take) are not the same. You can define as many constructors as you need. When a Java class contains multiple constructors, we say that the constructor is overloaded (comes in multiple

versions). This is what *constructor overloading* means, that a Java class contains multiple constructors.
Here is a Java constructor overloading example:

```
public class MyClass {

    private int number = 0;

    public MyClass() {
    }

    public MyClass(int theNumber) {
        this.number = theNumber;
    }
}
```

The Java class above contains two constructors. The first constructor is a no-arg constructor, meaning it takes no parameters (no arguments). The second constructor takes an `int` parameter. Inside the constructor body the `int` parameter value is assigned to a field, meaning the value of the parameter is copied into the field. The field is thus initialized to the given parameter value.
The keyword `this` in front of the field name (`this.number`) is not necessary. It just signals to the compiler that it is the field named `number` that is being referred to. This is explained in more detail in the section about constructor parameters.

Default, no-arg Constructor

You don't have to define a constructor for a class, but if you don't define any constructor, the Java compiler will insert a default, no-argument constructor for you. Thus, once the class is compiled it will always at least have a no-argument constructor.
If you do define a constructor for your class, then the Java compiler will not insert the default no-argument constructor into your class.

Constructor Parameters

As you have already seen, it is possible for a Java constructor to take parameters. These parameters can then be used to initialize the internal state (fields) of the newly created object. Here is an example:

```
public class Employee {

    private String firstName = null;
    private String lastName  = null;
    private int    birthYear = 0;


    public Employee(String first,
        String last,
        int    year   ) {

        firstName = first;
        lastName  = last;
        birthYear = year;
    }
}
```

In this example the Java constructor declaration is marked in bold. As you can see, three parameters are declared: `first`, `last` and `year`. Inside the body of the constructor the values of these three parameters are assigned to the fields of the `Employee` object.
The line breaks after each parameter are optional. The Java compiler ignores line breaks here. You can also write the parameter declaration in a single line if you want, like this:

```
public Employee(String first, String last, int year ) {
    firstName = first;
    lastName  = last;
    birthYear = year;
}
```

To call this constructor that takes three parameters, you would instantiate an `Employee` object like this:

```
Employee employee = new Employee("Jack", "Daniels", 2000);
```

The parameters are passed to the constructor inside the parentheses after the class name on the right side of the equal sign. The object is then created, and the constructor executed. After execution of the above constructor, the fields initialized by the constructor will have the values of the parameters passed to the constructor.
A Java constructor parameter can have the same name as a field. If a constructor parameter has the same name as a field, the Java compiler has problems knowing which you refer to. By default, if a parameter (or local variable) has the same name as a field in the same class, the parameter (or local variable) "shadows" for the field. Look at this constructor example:

```
public class Employee {

    private String firstName = null;
    private String lastName  = null;
    private int    birthYear = 0;


    public Employee(String firstName,
        String lastName,
        int    birthYear ) {

        firstName = firstName;
        lastName  = lastName;
        birthYear = birthYear;
    }

}
```

Inside the constructor of the Employee class the `firstName`, `lastName` and `birthYear` identifiers now refer to the constructor parameters, not to the Employee fields with the same

names. Thus, the constructor now just sets the parameters equal to themselves. The Employee fields are never initialized.
To signal to the Java compiler that you mean the fields of the Employee class and not the parameters, put the `this` keyword and a dot in front of the field name. Here is how the Java constructor declaration from before looks with that change:

```
public class Employee {

    private String firstName = null;
    private String lastName  = null;
    private int    birthYear = 0;


    public Employee(String firstName,
        String lastName,
        int    birthYear ) {

        this.firstName = firstName;
        this.lastName  = lastName;
        this.birthYear = birthYear;
    }

}
```

Now the Employee fields are correctly initialized inside the constructor.

Calling a Constructor

You call a constructor when you create a new instance of the class containing the constructor. Here is a Java constructor call example:

```
MyClass myClassVar = new MyClass();
```

This example invokes (calls) the no-argument constructor for `MyClass` as defined earlier in this text.
In case you want to pass parameters to the constructor, you include the parameters between the parentheses after the class name, like this:

```
MyClass myClassVar = new MyClass(1975);
```

This example passes one parameter to the `MyClass` constructor that takes an `int` as parameter.

Calling a Constructor From a Constructor

In Java it is possible to call a constructor from inside another constructor. When you call a constructor from inside another constructor, you use the `this` keyword to refer to the constructor. Here is an example of calling one constructor from within another constructor in Java:

```
public class Employee {

    private String firstName = null;
    private String lastName  = null;
    private int    birthYear = 0;

    public Employee(String first,
        String last,
        int    year   ) {

        firstName = first;
        lastName  = last;
        birthYear = year;
    }

    public Employee(String first, String last){
        this(first, last, -1);
    }
}
```

Notice the second constructor definition. Inside the body of the constructor you find this Java statement:

`this(first, last, -1);`

The `this` keyword followed by parentheses and parameters means that another constructor in the same Java class is being called. Which other constructor that is being called depends on what parameters you pass to the constructor call (inside the parentheses after the `this` keyword). In this example it is the first constructor in the class that is being called.

Calling Constructors in Superclasses

In Java a class can extend another class. When a class extends another class it is also said to "inherit" from the class it extends. The class that extends is called the subclass, and the class being extended is called the superclass.

A class that extends another class does not inherit its constructors. However, the subclass must call a constructor in the superclass inside one of the subclass constructors!

Look at the following two Java classes. The class Car extends (inherits from) the class Vehicle.

```
public class Vehicle {
    private String regNo = null;

    public Vehicle(String no) {
        this.regNo = no;
    }
}

public class Car extends Vehicle {
    private String brand = null;

    public Car(String br, String no) {
        super(no);
        this.brand = br;
    }
}
```

Notice the constructor in the Car class. It calls the constructor in the superclass using this Java statement:
```
super(no);
```

Using the keyword `super` refers to the superclass of the class using the `super` keyword. When `super` keyword is followed by parentheses like it is here, it refers to a constructor in the superclass. In this case it refers to the constructor in the Vehicle class. Because Car extends Vehicle, the Car constructors must all call a constructor in the Vehicle.
Java Constructor Access Modifiers
The access modifier of a constructor determines what classes in your application that are allowed to call that constructor.
For instance, if a constructor is declared `protected` then only classes in the same package, or subclasses of that class can call that constructor.
A class can have multiple constructors, and each constructor can have its own access modifier. Thus, some constructors may be available to all classes in your application, while other constructors are only available to classes in the same package, subclasses, or even only to the class itself (private constructors).

Throwing Exceptions From a Constructor

It is possible to throw an exception from a Java constructor. Here is an example of a Java class with a constructor that can throw an exception:
```
public class Car {

    public Car(String brand) throws Exception {
        if(brand == null) {
            throw new Exception("The brand parameter cannot be
null!");
        }
    }
}
```

Notice the `throws Exception` part of the constructor declaration. That part declares that the constructor may throw an `Exception`. If that happens, the created `Car` instance is not valid.
Here is an example of calling the `Car` constructor:
```
Car car = null;
try{
    car = new Car("Mercedes");
    //do something with car object
} catch(Exception e) {
    // handle exception
}
```

In case an exception is thrown from the `Car` constructor, the `car` variable will never be assigned a reference to the `Car` object you are trying to create. The `car` variable will still point to null.
Making a constructor throw an exception can be a good idea if you want to prevent an object of the given class to be created in an invalid state. Typically it is the input parameters to the

constructor that may cause the object to be invalid. If the constructor detects an invalid input parameter, it can throw an exception and prevent the assignment of the object to any variable.

Reference - http://tutorials.jenkov.com/java/constructors.html