# JAVA

# Core Java

## OOPs concepts in Java

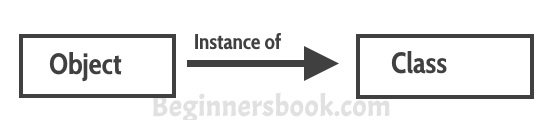
BY CHAITANYA SINGH | FILED UNDER: [OOPS CONCEPT](https://beginnersbook.com/category/oops-concept/)

Object-oriented programming System(OOPs) is a programming paradigm based on the concept of “objects” that contain data and methods. The primary purpose of object-oriented programming is to increase the flexibility and maintainability of programs. **Object oriented programming brings together data and its behaviour(methods) in a single location(object)** makes it easier to understand how a program works.

### OOPs Concepts – Table of Contents

1. [What is an Object](https://beginnersbook.com/2013/04/oops-concepts/#1)
2. [What is a class](https://beginnersbook.com/2013/04/oops-concepts/#2)
3. [Constructor in Java](https://beginnersbook.com/2013/04/oops-concepts/#3)
4. [Object Oriented Programming Features](https://beginnersbook.com/2013/04/oops-concepts/#4)
   * [Abstraction](https://beginnersbook.com/2013/04/oops-concepts/#5)
   * [Encapsulation](https://beginnersbook.com/2013/04/oops-concepts/#6)
   * [Inheritance](https://beginnersbook.com/2013/04/oops-concepts/#7)
   * [Polymorphism](https://beginnersbook.com/2013/04/oops-concepts/#8)
5. [Abstract Class and Methods](https://beginnersbook.com/2013/04/oops-concepts/#9)
6. [Interfaces in Java](https://beginnersbook.com/2013/04/oops-concepts/#10)

### What is an Object

  
**Object:** is a bundle of data and its behaviour(often known as methods).

Objects have two characteristics: They have **states** and **behaviors**.

**Examples of states and behaviors**  
**Example 1:**  
**Object**: House  
**State**: Address, Color, Area  
**Behavior**: Open door, close door

So if I had to write a class based on states and behaviours of House. I can do it like this: States can be represented as instance variables and behaviours as methods of the class. We will see how to create classes in the next section of this guide.

class House {

String address;

String color;

double area;

void openDoor() {

//Write code here

}

void closeDoor() {

//Write code here

}

...

...

}

**Example 2:**  
Let’s take another example.  
**Object**: Car  
**State**: Color, Brand, Weight, Model  
**Behavior**: Break, Accelerate, Slow Down, Gear change.

**Note:** As we have seen above, the states and behaviors of an object, can be represented by variables and methods in the class respectively.

### Characteristics of Objects:

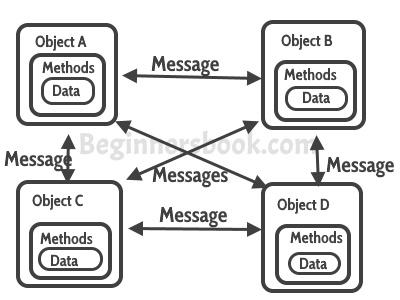
If you find it hard to understand Abstraction and Encapsulation, do not worry as I have covered these topics in detail with examples in the next section of this guide.

1. Abstraction
2. Encapsulation
3. Message passing

**Abstraction**: Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an **object** from the user.

**Encapsulation**: Encapsulation simply means **binding object state(fields) and behaviour(methods) together**. If you are **creating class**, you are doing encapsulation.

**Message passing**  
A single object by itself may not be very useful. An application contains many objects. One object interacts with another object by invoking methods on that object. It is also referred to as **Method Invocation**. See the diagram below.



### What is a Class in OOPs Concepts

A class can be considered as a blueprint using which you can create as many objects as you like. For example, **here we have a class Website that has two data members (also known as fields, instance variables and object states).** This is just a blueprint, it does not represent any website, however using this we can create Website objects (or instances) that represents the websites. We have created two objects, while creating objects we provided separate properties to the objects using constructor.

public class Website {

//fields (or instance variable)

String webName;

int webAge;

// constructor

Website(String name, int age){

this.webName = name;

this.webAge = age;

}

public static void main(String args[]){

//Creating objects

Website obj1 = new Website("beginnersbook", 5);

Website obj2 = new Website("google", 18);

//Accessing object data through reference

System.out.println(obj1.webName+" "+obj1.webAge);

System.out.println(obj2.webName+" "+obj2.webAge);

}

}

**Output:**

beginnersbook 5

google 18

### What is a Constructor

[Constructor](https://beginnersbook.com/2013/03/constructors-in-java/) looks like a method but it is in fact not a method. It’s name is same as class name and it does not return any value. You must have seen this statement in almost all the programs I have shared above:

MyClass obj = new MyClass();

If you look at the right side of this statement, we are calling the default constructor of class myClass to create a new object (or instance).

We can also have parameters in the constructor, such constructors are known as [parametrized constructors](https://beginnersbook.com/2014/01/parameterized-constructor-in-java-example/).

#### Example of constructor

public class ConstructorExample {

int age;

String name;

//**Default constructor**

ConstructorExample(){

this.name="Chaitanya";

this.age=30;

}

//**Parameterized constructor**

ConstructorExample(String n,int a){

this.name=n;

this.age=a;

}

public static void main(String args[]){

ConstructorExample obj1 = new ConstructorExample();

ConstructorExample obj2 =

new ConstructorExample("Steve", 56);

System.out.println(obj1.name+" "+obj1.age);

System.out.println(obj2.name+" "+obj2.age);

}

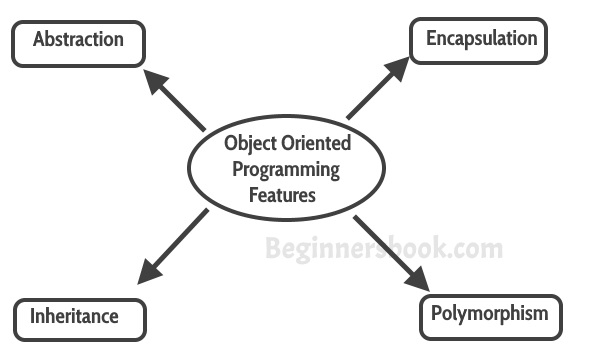
}

**Output:**

Chaitanya 30

Steve 56

### Object Oriented Programming features

  
These four features are the main OOPs Concepts that you must learn to understand the Object Oriented Programming in Java

### Abstraction

Abstraction is a process where you show only “**relevant” data and “hide” unnecessary details of an object** from the user. For example, when you login to your bank account online, you enter your user\_id and password and press login, what happens when you press login, how the input data sent to server, how it gets verified is all abstracted away from the you. Read more about it here: [Abstraction in Java](https://beginnersbook.com/2013/03/oops-in-java-encapsulation-inheritance-polymorphism-abstraction/).

### Encapsulation

Encapsulation simply means binding object state(fields) and behavior(methods) together. If you are creating class, you are doing encapsulation.

#### Encapsulation example in Java

How to  
1) **Make the instance variables private so that they cannot be accessed directly from outside the class**. You can only set and get values of these variables through the methods of the class.  
2) **Have getter and setter methods in the class to set and get the values** of the fields.

class EmployeeCount

{

private int numOfEmployees = 0;

public void setNoOfEmployees (int count)

{

numOfEmployees = count;

}

public double getNoOfEmployees ()

{

return numOfEmployees;

}

}

public class EncapsulationExample

{

public static void main(String args[])

{

EmployeeCount obj = new EmployeeCount ();

obj.setNoOfEmployees(5613);

System.out.println("No Of Employees: "+(int)obj.getNoOfEmployees());

}

}

**Output:**

No Of Employees: 5613

The class EncapsulationExample that is **using the Object of class EmployeeCount will not able to get the NoOfEmployees directly. It has to use the setter and getter methods of the same class to set and get the value**.  
**So what is the benefit of encapsulation in java programming**  
Well, at some point of time, if you want to change the implementation details of the class EmployeeCount, you can freely do so without affecting the classes that are using it.

### Inheritance

The process by which one class acquires the properties and functionalities of another class is called [inheritance](https://beginnersbook.com/2013/03/inheritance-in-java/). Inheritance provides the idea of reusability of code and each sub class defines only those features that are unique to it, rest of the features can be inherited from the parent class.

1. Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods.
2. Inheritance allows us to reuse of code, it improves reusability in your java application.
3. The parent class is called the **base class** or **super class**. The child class that extends the base class is called the derived class or **sub class** or **child class**.

**Note:** The biggest advantage of Inheritance is that the code in base class need not be rewritten in the child class.  
The **variables** and **methods** of the base class can be used in the **child class** as well.

#### Syntax: Inheritance in Java

To inherit a class we use extends keyword. Here class A is child class and class B is parent class.

class A extends B

{

}

#### Inheritance Example

In this example, we have a parent class Teacher and a child class MathTeacher. In the MathTeacher class we need not to write the same code which is already present in the present class. Here we have college name, designation and does() method that is common for all the teachers, thus MathTeacher class does not need to write this code, the common data members and methods can inherited from the Teacher class.

class Teacher {

String designation = "Teacher";

String college = "Beginnersbook";

void does(){

System.out.println("Teaching");

}

}

public class MathTeacher extends Teacher{

String mainSubject = "Maths";

public static void main(String args[]){

MathTeacher obj = new MathTeacher();

System.out.println(obj.college);

System.out.println(obj.designation);

System.out.println(obj.mainSubject);

obj.does();

}

}

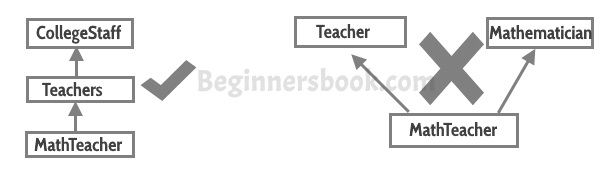
**Output:**

Beginnersbook

Teacher

Maths

Teaching

**Note: Multi-level inheritance** is allowed in Java but **not multiple inheritance**  


[Types of Inheritance](https://beginnersbook.com/2013/05/java-inheritance-types/):  
**Single Inheritance**: refers to a child and parent class relationship where a class extends the another class.

**Multilevel inheritance**: refers to a child and parent class relationship where a class extends the child class. For example class A extends class B and class B extends class C.

**Hierarchical inheritance**: refers to a child and parent class relationship where more than one classes extends the same class. For example, class B extends class A and class C extends class A.

**Multiple Inheritance**: refers to the concept of one class extending more than one classes, which means a child class has two parent classes. Java doesn’t support multiple inheritance, read more about it [here](https://beginnersbook.com/2013/05/java-multiple-inheritance/).

Most of the new **OO** **languages** like Small Talk, Java, C# do not support Multiple inheritance. Multiple Inheritance is supported in C++.

### Polymorphism

[Polymorphism](https://beginnersbook.com/2013/03/polymorphism-in-java/) is a object oriented programming feature that allows us to perform a single action in different ways. For example, lets say we have a class Animal that has a method animalSound(), here we cannot give implementation to this method as we do not know which Animal class would extend Animal class. So, we make this method abstract like this:

public abstract class Animal{

...

public abstract void animalSound();

}

Now suppose we have two Animal classes Dog and Lion that extends Animal class. We can provide the implementation detail there.

public class Lion extends Animal{

...

@Override

public void animalSound(){

System.out.println("Roar");

}

}

and

public class Dog extends Animal{

...

@Override

public void animalSound(){

System.out.println("Woof");

}

}

As you can see that although we had the common action for all subclasses animalSound() but there were different ways to do the same action. This is a perfect example of polymorphism (feature that allows us to perform a single action in different ways).

[Types of Polymorphism](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/)  
1) Static Polymorphism  
2) Dynamic Polymorphism

#### Static Polymorphism:

Polymorphism that is resolved during compiler time is known as static polymorphism. Method overloading can be considered as static polymorphism example.  
**Method Overloading**: This allows us to have more than one methods with same name in a class that differs in signature.

class DisplayOverloading

{

public void disp(char c)

{

System.out.println(c);

}

public void disp(char c, int num)

{

System.out.println(c + " "+num);

}

}

public class ExampleOverloading

{

public static void main(String args[])

{

DisplayOverloading obj = new DisplayOverloading();

obj.disp('a');

obj.disp('a',10);

}

}

**Output:**

a

a 10

When I say method signature I am not talking about return type of the method, for example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

#### Dynamic Polymorphism

It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime rather, thats why it is called runtime polymorphism.

**Example**

class Animal{

public void animalSound(){

System.out.println("Default Sound");

}

}

public class Dog extends Animal{

public void animalSound(){

System.out.println("Woof");

}

public static void main(String args[]){

Animal obj = new Dog();

obj.animalSound();

}

}

**Output:**

Woof

Since both the classes, child class and parent class have the same method animalSound. Which of the method will be called is determined at runtime by JVM.

**Few more overriding examples:**

Animal obj = new Animal();

obj.animalSound();

// This would call the Animal class method

Dog obj = new Dog();

obj.animalSound();

// This would call the Dog class method

Animal obj = new Dog();

obj.animalSound();

// This would call the Dog class method

#### IS-A & HAS-A Relationships

A Car **IS-A** Vehicle and **HAS-A** License then the code would look like this:

public class Vehicle{ }

public class Car extends Vehicle{

private License myCarLicense;

}

### Abstract Class and methods in OOPs Concepts

**Abstract method:**  
1) A method that is declared but not defined. Only method signature no body.  
2) Declared using the abstract keyword  
3) Example :

abstract public void playInstrument();

5) Used to put some kind of compulsion on the class who inherits the class has abstract methods. The class that inherits must provide the implementation of all the abstract methods of parent class else declare the **subclass as abstract**.  
6) These cannot be abstract

* Constructors
* Static methods
* Private methods
* Methods that are declared “final”

**Abstract Class**  
An abstract .

abstract class A{

abstract void myMethod();

void anotherMethod(){

//Does something

}

}

**Note 1:** There can be some scenarios where it is difficult to implement all the methods in the base class. In such scenarios one can define the base class as an abstract class which signifies that this base class is a special kind of class which is not complete on its own.

A class derived from the abstract base class must implement those methods that are not implemented(means they are abstract) in the abstract class.

**Note 2:** Abstract class cannot be instantiated which means you cannot create the object of abstract class. To use this class, you need to create another class that extends this abstract class provides the implementation of abstract methods, then you can use the object of that child class to call non-abstract parent class methods as well as implemented methods(those that were abstract in parent but implemented in child class).

**Note 3:** If a child does not implement all the abstract methods of parent class(the abstract class), then the child class must need to be declared abstract.

### Example of Abstract class and Methods

Here we have an abstract class Animal that has an abstract method animalSound(), since the animal sound differs from one animal to another, there is no point in giving the implementation to this method as every child class must override this method to give its own implementation details. That’s why we made it abstract.  
Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method. This way we ensures that every animal has a sound.

//abstract class

abstract class Animal{

//abstract method

public abstract void animalSound();

}

public class Dog extends Animal{

public void animalSound(){

System.out.println("Woof");

}

public static void main(String args[]){

Animal obj = new Dog();

obj.animalSound();

}

}

Output:

Woof

### Interfaces in Java

**An interface is a blueprint of a class**, which can be declared by using **interface** keyword. Interfaces can contain only constants and abstract methods (methods with only signatures no body).Like abstract classes**, Interfaces cannot be instantiated, they can only be implemented by classes or extended by other interfaces. Interface is a common way to achieve full abstraction in Java.**

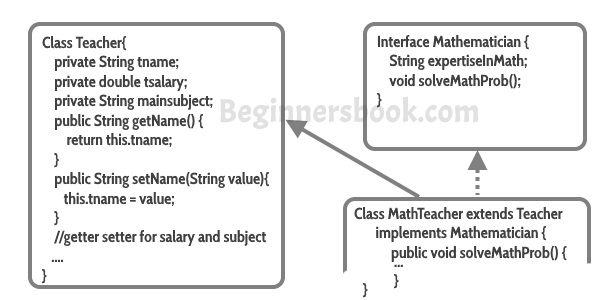
**Note:**

1. Java does not support Multiple Inheritance, however a **class can implement more than one interfaces**
2. **Interface is similar to an abstract class but it contains only abstract methods.**
3. Interfaces are created by using interface keyword instead of the keyword class
4. We use implements keyword while implementing an interface(similar to extending a class with extends keyword)

**Interface: Syntax**

class ClassName extends Superclass implements Interface1, Interface2, ....

### Example of Interface:



**Note**:

1. All **methods in an interface** are implicitly public and abstract. Using the keyword abstract before each method is optional.
2. An **interface** may contain final variables.
3. A class can **extend only one other class**, but it can **implement any number of interfaces.**
4. **When a class implements an interface it has to give the definition of all the abstract methods of interface, else it can be declared as abstract class**
5. An interface reference can point to **objects** of its implementing classes.

**Generalization and Specialization:**  
In order to implement the concept of inheritance in an OOPs, one has to first identify the similarities among different classes so as to come up with the base class.

This process of identifying the similarities among different classes is called **Generalization. Generalization is the process of extracting shared characteristics** from two or more classes, and combining them into a generalized superclass. Shared characteristics can be attributes or methods.

In contrast to generalization, specialization means creating new subclasses from an existing class. If it turns out that certain attributes or methods only apply to some of the objects of the class, a subclass can be created.

#### Access Specifiers

Well, you must have seen public, private keyword in the examples I have shared above. They are called [access specifiers](https://beginnersbook.com/2013/05/java-access-modifiers/) as they decide the scope of a data member, method or class.

There are **four types** of access specifiers in java:  
**public:** **Accessible to all**. Other objects can also access this member variable or function.  
**private:** Not accessible by other objects. Private members can be accessed only by the methods in the same class. **Object accessible only in class in which they are declared.**  
**protected:** The scope of a protected variable is within the class which declares it and in the class which inherits from the class (Scope is class and subclass).  
**Default:** Scope is Package Level. We do not need to explicitly mention default as when we do not mention any access specifier it is considered as default.

## Comparable and Comparator in Java

<https://www.journaldev.com/780/comparable-and-comparator-in-java-example>

Comparable and Comparator in Java are very useful for sorting the collection of objects. Java provides some inbuilt methods to sort primitive types array or Wrapper classes array or list. Here we will first learn how we can sort an array/list of primitive types and wrapper classes and then we will use **java.lang.Comparable** and **java.util.Comparator** interfaces to sort array/list of custom classes.

Let’s see how we can sort primitive types or Object array and list with a simple program.

package com.journaldev.sort;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public class JavaObjectSorting {

/\*\*

\* This class shows how to sort primitive arrays,

\* Wrapper classes Object Arrays

\* @param args

\*/

public static void main(String[] args) {

//sort primitives array like int array

int[] intArr = {5,9,1,10};

Arrays.sort(intArr);

System.out.println(Arrays.toString(intArr));

//sorting String array

String[] strArr = {"A", "C", "B", "Z", "E"};

Arrays.sort(strArr);

System.out.println(Arrays.toString(strArr));

//sorting list of objects of Wrapper classes

List<String> strList = new ArrayList<String>();

strList.add("A");

strList.add("C");

strList.add("B");

strList.add("Z");

strList.add("E");

Collections.sort(strList);

for(String str: strList) System.out.print(" "+str);

}

}

Output of the above program is:

[1, 5, 9, 10]

[A, B, C, E, Z]

A B C E Z

Now let’s try to sort an array of objects.

package com.journaldev.sort;

public class Employee {

private int id;

private String name;

private int age;

private long salary;

public int getId() {

return id;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

public long getSalary() {

return salary;

}

public Employee(int id, String name, int age, int salary) {

this.id = id;

this.name = name;

this.age = age;

this.salary = salary;

}

@Override

//this is overridden to print the user-friendly information about the Employee

public String toString() {

return "[id=" + this.id + ", name=" + this.name + ", age=" + this.age + ", salary=" +

this.salary + "]";

}

}

Here is the code I used to sort the array of Employee objects.

//sorting object array

Employee[] empArr = new Employee[4];

empArr[0] = new Employee(10, "Mikey", 25, 10000);

empArr[1] = new Employee(20, "Arun", 29, 20000);

empArr[2] = new Employee(5, "Lisa", 35, 5000);

empArr[3] = new Employee(1, "Pankaj", 32, 50000);

//sorting employees array using Comparable interface implementation

Arrays.sort(empArr);

System.out.println("Default Sorting of Employees list:\n"+Arrays.toString(empArr));

When I tried to run this, it throws the following runtime exception.

Exception in thread "main" java.lang.ClassCastException: com.journaldev.sort.Employee cannot be cast to java.lang.Comparable

at java.util.ComparableTimSort.countRunAndMakeAscending(ComparableTimSort.java:290)

at java.util.ComparableTimSort.sort(ComparableTimSort.java:157)

at java.util.ComparableTimSort.sort(ComparableTimSort.java:146)

at java.util.Arrays.sort(Arrays.java:472)

at com.journaldev.sort.JavaSorting.main(JavaSorting.java:41)

**Table of Contents**[[hide](https://www.journaldev.com/780/comparable-and-comparator-in-java-example)]

* [1 Comparable and Comparator](https://www.journaldev.com/780/comparable-and-comparator-in-java-example#comparable-and-comparator)
* [2 Java Comparator](https://www.journaldev.com/780/comparable-and-comparator-in-java-example#java-comparator)
* [3 Comparable vs Comparator](https://www.journaldev.com/780/comparable-and-comparator-in-java-example#comparable-vs-comparator)

## Comparable and Comparator

[](https://cdn.journaldev.com/wp-content/uploads/2012/11/comparable-and-comparator-in-java.jpg)

Java provides **Comparable** interface which should be implemented by any custom class if we want to use [Arrays](https://www.journaldev.com/16770/java-arrays-java-util-arrays) or [Collections](https://www.journaldev.com/16635/collections-class-java-util-collections) sorting methods.

The Comparable interface has **compareTo(T obj)** method which is used by sorting methods, you can check any Wrapper, String or Date class to confirm this. We should override this method in such a way that it returns a negative integer, zero, or a positive integer if “this” object is less than, equal to, or greater than the object passed as an argument.

After implementing Comparable [interface](https://www.journaldev.com/1601/interface-in-java) in Employee class, here is the resulting Employee class.

package com.journaldev.sort;

import java.util.Comparator;

public class Employee implements Comparable<Employee> {

private int id;

private String name;

private int age;

private long salary;

public int getId() {

return id;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

public long getSalary() {

return salary;

}

public Employee(int id, String name, int age, int salary) {

this.id = id;

this.name = name;

this.age = age;

this.salary = salary;

}

@Override

public int compareTo(Employee emp) {

//let's sort the employee based on an id in ascending order

//returns a negative integer, zero, or a positive integer as this employee id

//is less than, equal to, or greater than the specified object.

return (this.id - emp.id);

}

@Override

//this is required to print the user-friendly information about the Employee

public String toString() {

return "[id=" + this.id + ", name=" + this.name + ", age=" + this.age + ", salary=" +

this.salary + "]";

}

}

Now when we execute the above snippet for Arrays sorting of Employees and print it, here is the output.

Default Sorting of Employees list:

[[id=1, name=Pankaj, age=32, salary=50000], [id=5, name=Lisa, age=35, salary=5000], [id=10, name=Mikey, age=25, salary=10000], [id=20, name=Arun, age=29, salary=20000]]

As you can see that Employees array is sorted by id in ascending order.

But, in most real-life scenarios, we want sorting based on different parameters. For example, as a CEO, I would like to sort the employees based on Salary, an HR would like to sort them based on age. This is the situation where we need to use **Java Comparator** interface because Comparable.compareTo(Object o) method implementation can provide default sorting and we can’t change it dynamically. Whereas with Comparator, we can define multiple methods with different ways of sorting and then chose the sorting method based on our requirements.

## Java Comparator

Comparator interface compare(Object o1, Object o2) method need to be implemented that takes two Object argument, it should be implemented in such a way that it returns negative int if the first argument is less than the second one and returns zero if they are equal and positive int if the first argument is greater than the second one.

Comparable and Comparator interfaces use [Generics](https://www.journaldev.com/1663/java-generics-example-method-class-interface) for compile-time type checking, learn more about [Java Generics](https://www.journaldev.com/1663/java-generics-example-method-class-interface).

Here is how we can create different Comparator implementation in the Employee class.

/\*\*

\* Comparator to sort employees list or array in order of Salary

\*/

public static Comparator<Employee> SalaryComparator = new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return (int) (e1.getSalary() - e2.getSalary());

}

};

/\*\*

\* Comparator to sort employees list or array in order of Age

\*/

public static Comparator<Employee> AgeComparator = new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return e1.getAge() - e2.getAge();

}

};

/\*\*

\* Comparator to sort employees list or array in order of Name

\*/

public static Comparator<Employee> NameComparator = new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return e1.getName().compareTo(e2.getName());

}

};

All the above implementations of Comparator interface are [**anonymous classes**](https://www.journaldev.com/996/java-inner-class).

We can use these comparators to pass an argument to sort function of Arrays and Collections classes.

//sort employees array using Comparator by Salary

Arrays.sort(empArr, Employee.SalaryComparator);

System.out.println("Employees list sorted by Salary:\n"+Arrays.toString(empArr));

//sort employees array using Comparator by Age

Arrays.sort(empArr, Employee.AgeComparator);

System.out.println("Employees list sorted by Age:\n"+Arrays.toString(empArr));

//sort employees array using Comparator by Name

Arrays.sort(empArr, Employee.NameComparator);

System.out.println("Employees list sorted by Name:\n"+Arrays.toString(empArr));

Here is the output of the above code snippet:

Employees list sorted by Salary:

[[id=5, name=Lisa, age=35, salary=5000], [id=10, name=Mikey, age=25, salary=10000], [id=20, name=Arun, age=29, salary=20000], [id=1, name=Pankaj, age=32, salary=50000]]

Employees list sorted by Age:

[[id=10, name=Mikey, age=25, salary=10000], [id=20, name=Arun, age=29, salary=20000], [id=1, name=Pankaj, age=32, salary=50000], [id=5, name=Lisa, age=35, salary=5000]]

Employees list sorted by Name:

[[id=20, name=Arun, age=29, salary=20000], [id=5, name=Lisa, age=35, salary=5000], [id=10, name=Mikey, age=25, salary=10000], [id=1, name=Pankaj, age=32, salary=50000]]

So now we know that if we want to sort java object array or list, we need to implement java Comparable interface to provide default sorting and we should implement java Comparator interface to provide different ways of sorting.

We can also create separate class that implements Comparator interface and then use it.

Here is the final classes we have explaining **Comparable and Comparator** in Java.

package com.journaldev.sort;

import java.util.Comparator;

public class Employee implements Comparable<Employee> {

private int id;

private String name;

private int age;

private long salary;

public int getId() {

return id;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

public long getSalary() {

return salary;

}

public Employee(int id, String name, int age, int salary) {

this.id = id;

this.name = name;

this.age = age;

this.salary = salary;

}

@Override

public int compareTo(Employee emp) {

//let's sort the employee based on an id in ascending order

//returns a negative integer, zero, or a positive integer as this employee id

//is less than, equal to, or greater than the specified object.

return (this.id - emp.id);

}

@Override

//this is required to print the user-friendly information about the Employee

public String toString() {

return "[id=" + this.id + ", name=" + this.name + ", age=" + this.age + ", salary=" +

this.salary + "]";

}

/\*\*

\* Comparator to sort employees list or array in order of Salary

\*/

public static Comparator<Employee> SalaryComparator = new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return (int) (e1.getSalary() - e2.getSalary());

}

};

/\*\*

\* Comparator to sort employees list or array in order of Age

\*/

public static Comparator<Employee> AgeComparator = new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return e1.getAge() - e2.getAge();

}

};

/\*\*

\* Comparator to sort employees list or array in order of Name

\*/

public static Comparator<Employee> NameComparator = new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return e1.getName().compareTo(e2.getName());

}

};

}

Here is the separate class implementation of Comparator interface that will compare two Employees object first on their id and if they are same then on the name.

package com.journaldev.sort;

import java.util.Comparator;

public class EmployeeComparatorByIdAndName implements Comparator<Employee> {

@Override

public int compare(Employee o1, Employee o2) {

int flag = o1.getId() - o2.getId();

if(flag==0) flag = o1.getName().compareTo(o2.getName());

return flag;

}

}

Here is the test class where we are using different ways to sort Objects in java using Comparable and Comparator.

package com.journaldev.sort;

import java.util.Arrays;

public class JavaObjectSorting {

/\*\*

\* This class shows how to sort custom objects array/list

\* implementing Comparable and Comparator interfaces

\* @param args

\*/

public static void main(String[] args) {

//sorting custom object array

Employee[] empArr = new Employee[4];

empArr[0] = new Employee(10, "Mikey", 25, 10000);

empArr[1] = new Employee(20, "Arun", 29, 20000);

empArr[2] = new Employee(5, "Lisa", 35, 5000);

empArr[3] = new Employee(1, "Pankaj", 32, 50000);

//sorting employees array using Comparable interface implementation

Arrays.sort(empArr);

System.out.println("Default Sorting of Employees list:\n"+Arrays.toString(empArr));

//sort employees array using Comparator by Salary

Arrays.sort(empArr, Employee.SalaryComparator);

System.out.println("Employees list sorted by Salary:\n"+Arrays.toString(empArr));

//sort employees array using Comparator by Age

Arrays.sort(empArr, Employee.AgeComparator);

System.out.println("Employees list sorted by Age:\n"+Arrays.toString(empArr));

//sort employees array using Comparator by Name

Arrays.sort(empArr, Employee.NameComparator);

System.out.println("Employees list sorted by Name:\n"+Arrays.toString(empArr));

//Employees list sorted by ID and then name using Comparator class

empArr[0] = new Employee(1, "Mikey", 25, 10000);

Arrays.sort(empArr, new EmployeeComparatorByIdAndName());

System.out.println("Employees list sorted by ID and Name:\n"+Arrays.toString(empArr));

}

}

Here is the output of the above program:

Default Sorting of Employees list:

[[id=1, name=Pankaj, age=32, salary=50000], [id=5, name=Lisa, age=35, salary=5000], [id=10, name=Mikey, age=25, salary=10000], [id=20, name=Arun, age=29, salary=20000]]

Employees list sorted by Salary:

[[id=5, name=Lisa, age=35, salary=5000], [id=10, name=Mikey, age=25, salary=10000], [id=20, name=Arun, age=29, salary=20000], [id=1, name=Pankaj, age=32, salary=50000]]

Employees list sorted by Age:

[[id=10, name=Mikey, age=25, salary=10000], [id=20, name=Arun, age=29, salary=20000], [id=1, name=Pankaj, age=32, salary=50000], [id=5, name=Lisa, age=35, salary=5000]]

Employees list sorted by Name:

[[id=20, name=Arun, age=29, salary=20000], [id=5, name=Lisa, age=35, salary=5000], [id=10, name=Mikey, age=25, salary=10000], [id=1, name=Pankaj, age=32, salary=50000]]

Employees list sorted by ID and Name:

[[id=1, name=Mikey, age=25, salary=10000], [id=1, name=Pankaj, age=32, salary=50000], [id=5, name=Lisa, age=35, salary=5000], [id=10, name=Mikey, age=25, salary=10000]]

The **java.lang.Comparable** and **java.util.Comparator** are powerful interfaces that can be used to provide sorting objects in java.

## Comparable vs Comparator

1. Comparable interface can be used to provide single way of sorting whereas Comparator interface is used to provide different ways of sorting.
2. For using Comparable, Class needs to implement it whereas for using Comparator we don’t need to make any change in the class.
3. Comparable interface is in java.lang package whereas Comparator interface is present in java.util package.
4. We don’t need to make any code changes at client side for using Comparable, Arrays.sort() or Collection.sort() methods automatically uses the compareTo() method of the class. For Comparator, client needs to provide the Comparator class to use in compare() method.

# [Why String is Immutable or Final in Java](https://javarevisited.blogspot.com/2010/10/why-string-is-immutable-or-final-in-java.html)

The string is Immutable in Java because String objects are cached in String pool. Since cached String literals are shared between multiple clients there is always a risk, where one client's action would affect all another client. For example, if one client changes the value of String "Test" to "TEST", all other clients will also see that value as explained in the first example. Since caching of String objects was important from performance reason this risk was avoided by making String class Immutable. At the same time, [*String was made final*](http://java67.blogspot.com/2014/01/why-string-class-has-made-immutable-or-final-java.html) so that no one can compromise invariant of String class e.g. Immutability, Caching, hashcode calculation etc by extending and overriding behaviors. Another reason of *why String class is immutable* could die due to HashMap.

Since Strings are very popular as HashMap key, it's important for them to be immutable so that they can retrieve the value object which was stored in HashMap. Since [HashMap works in the principle of hashing](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html), which requires same has value to function properly. Mutable String would produce two different hashcodes at the time of insertion and retrieval if contents of String was modified after insertion, potentially losing the value object in the map.  
  
If you are an Indian cricket fan, you may be able to correlate with my next sentence. The string is VVS Laxman of Java, i.e. very very special class. I have not seen a single Java program which is written without using String. That's why a solid understanding of String is very important for a Java developer.

## Why String is Immutable in Java

**·**

This is an old yet still popular question. There are multiple reasons that String is designed to be immutable in Java. A good answer depends on good understanding of memory, synchronization, data structures, etc. In the following, I will summarize some answers.

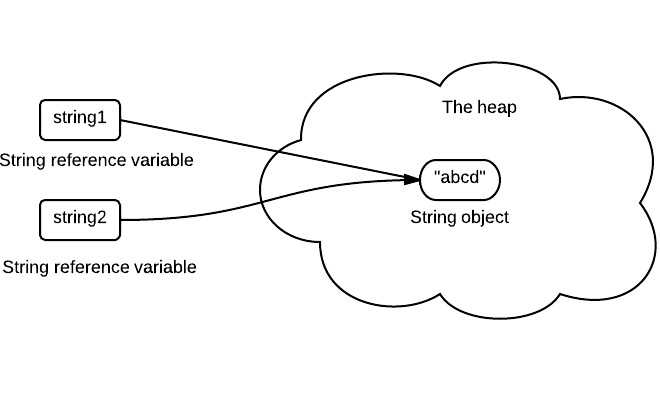
**1. Requirement of String Pool**

String pool (String intern pool) is a special storage area in Java heap. When a string is created and if the string already exists in the pool, the reference of the existing string will be returned, instead of creating a new object and returning its reference.

The following code will create only one string object in the heap.

String string1 = "abcd";

String string2 = "abcd";

Here is how it looks:  
[](http://www.programcreek.com/wp-content/uploads/2013/07/java-string-pool.jpeg)

If string is not immutable, changing the string with one reference will lead to the wrong value for the other references.

**2. Allow String to Cache its Hashcode**

The hashcode of string is frequently used in Java. For example, in a HashMap. Being immutable guarantees that hashcode will always the same, so that it can be cashed without worrying the changes.That means, there is no need to calculate hashcode every time it is used. This is more efficient.

**3. Security**

String is widely used as parameter for many java classes, e.g. **network connection, opening files, etc. Were String not immutable, a connection or file would be changed and lead to serious security threat**. The method thought it was connecting to one machine, but was not. Mutable strings could cause security problem in Reflection too, as the parameters are strings.

Here is a code example:

boolean connect(string s){

if (!isSecure(s)) {

throw new SecurityException();

}

//here will cause problem, if s is changed before this by using other references.

causeProblem(s);

}

## difference of print and Println in Java?

**Difference** between the methods **print and println**. The **println**("...") method **prints** the string "..." and moves the cursor to a new line. The **print**("...") method instead **prints** just the string "...", but does not move the cursor to a new line. Hence, subsequent **printing** instructions will **print** on the same line.

## Java String join() with examples

The **java.lang.string.join()** method concatenates the given elements with the delimiter and returns the concatenated string.Note that if an element is null, then null is added.The **join()** method is included in java string since JDK 1.8.  
There are two types of **join()** methods in java string.  
**Syntax:**

**public static String join(CharSequence deli, CharSequence... ele)**

and

public static String **join**

**(CharSequence deli, Iterable<? extends CharSequence> ele)**

**Parameters:**

**deli**- delimiter to be attached with each element

**ele**- string or char to be attached with delimiter

**Returns :**  string joined with delimiter.

|  |
| --- |
| // Java program to demonstrate  // working of join() method    class Gfg1 {      public static void main(String args[])      {          // delimiter is "<" and elements are "Four", "Five", "Six", "Seven"          String gfg1 = String.join(" < ", "Four", "Five", "Six", "Seven");            System.out.println(gfg1);      }  } |

**Output:**

Four < Five < Six < Seven

|  |
| --- |
| // Java program to demonstrate  // working of join() method    class Gfg2 {      public static void main(String args[])      {          // delimiter is "  " and elements are "My",          // "name", "is", "Niraj", "Pandey"          String gfg2 = String.join("  ", "My", "name", "is", "Niraj", "Pandey");            System.out.println(gfg2);      }  } |

**Output:**

My name is Niraj Pandey

|  |
| --- |
| // Java program to demonstrate  // working of join() method    class Gfg3 {      public static void main(String args[])      {          // delimiter is "->" and elements are "Wake up",          // "Eat", "Play", "Sleep", "Wake up"            String gfg3 = String.join("-> ", "Wake up", "Eat",                        "Play", "Sleep", "Wake up");            System.out.println(gfg3);      }  } |

**Output:**

Wake up-> Eat-> Play-> Sleep-> Wake up

## Difference between StringBuffer and StringBuilder

Java provides three classes to represent a sequence of characters: String, StringBuffer, and StringBuilder. The String class is an immutable class whereas StringBuffer and StringBuilder classes are mutable. There are many differences between StringBuffer and StringBuilder. The StringBuilder class is introduced since JDK 1.5.

A list of differences between StringBuffer and StringBuilder are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **StringBuffer** | **StringBuilder** |
| 1) | StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| 2) | StringBuffer is *less efficient* than StringBuilder. | StringBuilder is *more efficient* than StringBuffer. |

## [What is the difference between concat and append?](http://erpbasic.blogspot.com/2012/02/what-is-difference-between-concat-and.html)

Ans:

1. Concat is used to add a String at the end of another String. Append adds a String          or character sequence to StringBufffer.

2. Concat creates a new String object, whereas StringBuffer append doesn't.

3. Append is more efficient than concat

## StringBuffer toString() method in Java with Examples

The **toString()** method of **StringBuffer class** is the inbuilt method used to returns a string representing the data contained by StringBuffer Object. A new String object is created and initialized to get the character sequence from this StringBuffer object and then String is returned by toString(). Subsequent changes to this sequence contained by Object do not affect the contents of the String.

**Syntax:**

public abstract String toString()

**Return Value:** This method returns the **String representing the data contained by StringBuffer Object.**

**Below programs illustrate the StringBuffer.toString() method:**

|  |
| --- |
| // Java program to demonstrate  // the toString() Method.    class GFG {      public static void main(String[] args)      {            // create a StringBuffer object          // with a String pass as parameter          StringBuffer str              = new StringBuffer("GeeksForGeeks");            // print string          System.out.println("String contains = "                             + str.toString());      }  } |

**Output:**

**String contains = GeeksForGeeks**

## JAVA String imp methods:

### String replaceAll()

The **java string replaceAll()** method returns a string replacing all the sequence of characters matching regex and replacement string.

#### Signature

1. **public** String replaceAll(String regex, String replacement)

#### Parameters

**regex** : regular expression

**replacement** : replacement sequence of characters

#### Returns

replaced string

#### Java String replaceAll() example: replace character

Let's see an example to replace all the occurrences of **a single character**.

1. **public** **class** ReplaceAllExample1{
2. **public** **static** **void** main(String args[]){
3. String s1="javatpoint is a very good website";
4. String replaceString=s1.replaceAll("a","e");//replaces all occurrences of "a" to "e"
5. System.out.println(replaceString);
6. }}

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=ReplaceAllExample1)

jevetpoint is e very good website

#### Java String replaceAll() example: replace word

Let's see an example to replace all the occurrences of **single word or set of words**.

1. **public** **class** ReplaceAllExample2{
2. **public** **static** **void** main(String args[]){
3. String s1="My name is Khan. My name is Bob. My name is Sonoo.";
4. String replaceString=s1.replaceAll("is","was");//replaces all occurrences of "is" to "was"
5. System.out.println(replaceString);
6. }}

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=ReplaceAllExample2)

My name was Khan. My name was Bob. My name was Sonoo.

#### Java String replaceAll() example: remove white spaces

Let's see an example to remove all the occurrences of **white spaces**.

1. **public** **class** ReplaceAllExample3{
2. **public** **static** **void** main(String args[]){
3. String s1="My name is Khan. My name is Bob. My name is Sonoo.";
4. String replaceString=s1.replaceAll("\\s","");
5. System.out.println(replaceString);
6. }}

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=ReplaceAllExample3)

MynameisKhan.MynameisBob.MynameisSonoo.

### Substring in Java

A part of string is called **substring**. In other words, substring is a subset of another string. In case of substring startIndex is inclusive and endIndex is exclusive.

#### Example of java substring

1. **public** **class** TestSubstring{
2. **public** **static** **void** main(String args[]){
3. String s="SachinTendulkar";
4. System.out.println(s.substring(6));//Tendulkar
5. System.out.println(s.substring(0,6));//Sachin
6. }
7. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSubstring)

Tendulkar

Sachin

### Java String charAt()

The **java string charAt()** method returns a char value at the given index number.

The index number starts from 0 and goes to n-1, where n is length of the string. It returns **StringIndexOutOfBoundsException** if given index number is greater than or equal to this string length or a negative number.

#### Signature

The signature of string charAt() method is given below:

1. **public** **char** charAt(**int** index)

#### Parameter

**index** : index number, starts with 0

#### Returns

**A char value**

#### Specified by

**CharSequence** interface, located inside java.lang package.

#### Throws

**StringIndexOutOfBoundsException** : if index is negative value or greater than this string length.

#### Java String charAt() method example

1. **public** **class** CharAtExample{
2. **public** **static** **void** main(String args[]){
3. String name="javatpoint";
4. **char** ch=name.charAt(4);//returns the char value at the 4th index
5. System.out.println(ch);
6. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=CharAtExample)

Output:

t

#### StringIndexOutOfBoundsException with charAt()

Let's see the example of charAt() method where we are passing greater index value. In such case, it throws StringIndexOutOfBoundsException at run time.

1. **public** **class** CharAtExample{
2. **public** **static** **void** main(String args[]){
3. String name="javatpoint";
4. **char** ch=name.charAt(10);//returns the char value at the 10th index
5. System.out.println(ch);
6. }}

Output:

Exception in thread "main" java.lang.StringIndexOutOfBoundsException:

String index out of range: 10

at java.lang.String.charAt(String.java:658)

at CharAtExample.main(CharAtExample.java:4)

#### Java String charAt() Example 3

Let's see a simple example where we are accessing first and last character from the provided string.

1. **public** **class** CharAtExample3 {
2. **public** **static** **void** main(String[] args) {
3. String str = "Welcome to Javatpoint portal";
4. **int** strLength = str.length();
5. // Fetching first character
6. System.out.println("Character at 0 index is: "+ str.charAt(0));
7. // The last Character is present at the string length-1 index
8. System.out.println("Character at last index is: "+ str.charAt(strLength-1));
9. }
10. }

Output:

Character at 0 index is: W

Character at last index is: l

#### Java String charAt() Example 4

Let's see an example where we are accessing all the elements present at odd index.

1. **public** **class** CharAtExample4 {
2. **public** **static** **void** main(String[] args) {
3. String str = "Welcome to Javatpoint portal";
4. **for** (**int** i=0; i<=str.length()-1; i++) {
5. **if**(i%2!=0) {
6. System.out.println("Char at "+i+" place "+str.charAt(i));
7. }
8. }
9. }
10. }

Output:

Char at 1 place e

Char at 3 place c

Char at 5 place m

Char at 7 place

Char at 9 place o

Char at 11 place J

Char at 13 place v

Char at 15 place t

Char at 17 place o

Char at 19 place n

Char at 21 place

Char at 23 place o

Char at 25 place t

Char at 27 place l

#### Java String charAt() Example 5

Let's see an example where we are counting frequency of a character in the string.

1. **public** **class** CharAtExample5 {
2. **public** **static** **void** main(String[] args) {
3. String str = "Welcome to Javatpoint portal";
4. **int** count = 0;
5. **for** (**int** i=0; i<=str.length()-1; i++) {
6. **if**(str.charAt(i) == 't') {
7. count++;
8. }
9. }
10. System.out.println("Frequency of t is: "+count);
11. }
12. }

Output:

Frequency of t is: 4

### Java String compareTo() method example

1. **public** **class** CompareToExample{
2. **public** **static** **void** main(String args[]){
3. String s1="hello";
4. String s2="hello";
5. String s3="meklo";
6. String s4="hemlo";
7. String s5="flag";
8. System.out.println(s1.compareTo(s2));//0 because both are equal
9. System.out.println(s1.compareTo(s3));//-5 because "h" is 5 times lower than "m"
10. System.out.println(s1.compareTo(s4));//-1 because "l" is 1 times lower than "m"
11. System.out.println(s1.compareTo(s5));//2 because "h" is 2 times greater than "f"

### Java String concat

The **java string concat()** method combines specified string at the end of this string. It returns combined string. It is like appending another string.

#### Signature

The signature of string concat() method is given below:

1. **public** String concat(String anotherString)

#### Parameter

**anotherString** : another string i.e. to be combined at the end of this string.

#### Returns

combined string

#### Java String concat() method example

1. **public** **class** ConcatExample{
2. **public** **static** **void** main(String args[]){
3. String s1="java string";
4. s1.concat("is immutable");
5. System.out.println(s1);
6. s1=s1.concat(" is immutable so assign it explicitly");
7. System.out.println(s1);
8. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=ConcatExample)

java string

java string is immutable so assign it explicitly

#### Java String concat() Method Example 2

Let's see an example where we are concatenating multiple string objects.

1. **public** **class** ConcatExample2 {
2. **public** **static** **void** main(String[] args) {
3. String str1 = "Hello";
4. String str2 = "Javatpoint";
5. String str3 = "Reader";
6. // Concatenating one string
7. String str4 = str1.concat(str2);
8. System.out.println(str4);
9. // Concatenating multiple strings
10. String str5 = str1.concat(str2).concat(str3);
11. System.out.println(str5);
12. }
13. }

Output:

HelloJavatpoint

HelloJavatpointReader

#### Java String concat() Method Example 3

Let's see an example where we are concatenating spaces and special chars to the string object.

1. **public** **class** ConcatExample3 {
2. **public** **static** **void** main(String[] args) {
3. String str1 = "Hello";
4. String str2 = "Javatpoint";
5. String str3 = "Reader";
6. // Concatenating Space among strings
7. String str4 = str1.concat(" ").concat(str2).concat(" ").concat(str3);
8. System.out.println(str4);
9. // Concatenating Special Chars
10. String str5 = str1.concat("!!!");
11. System.out.println(str5);
12. String str6 = str1.concat("@").concat(str2);
13. System.out.println(str6);
14. }
15. }

Output:

Hello Javatpoint Reader

Hello!!!

Hello@Javatpoint

### Java String contains() Method Example 3

The contains() method is helpful to find a char-sequence in the string. We can use it in control structure to produce search based result. Let us see an example below.

1. **public** **class** ContainsExample3 {
2. **public** **static** **void** main(String[] args) {
3. String str = "To learn Java visit Javatpoint.com";
4. **if**(str.contains("Javatpoint.com")) {
5. System.out.println("This string contains javatpoint.com");
6. }**else**
7. System.out.println("Result not found");
8. }
9. }

Output:

This string contains javatpoint.com

### Java String endsWith() Method Example 2

1. **public** **class** EndsWithExample2 {
2. **public** **static** **void** main(String[] args) {
3. String str = "Welcome to Javatpoint.com";
4. System.out.println(str.endsWith("point"));
5. **if**(str.endsWith(".com")) {
6. System.out.println("String ends with .com");
7. }**else** System.out.println("It does not end with .com");
8. }
9. }

Output:

false

String ends with .com

### Java String equals() Method Example 2

The equals() method compares two strings and can be used in if-else control structure.

1. **public** **class** EqualsExample {
2. **public** **static** **void** main(String[] args) {
3. String s1 = "javatpoint";
4. String s2 = "javatpoint";
5. String s3 = "Javatpoint";
6. System.out.println(s1.equals(s2)); // True because content is same
7. **if** (s1.equals(s3)) {
8. System.out.println("both strings are equal");
9. }**else** System.out.println("both strings are unequal");
10. }
11. }

true

both strings are unequal

#### Java String equals() Method Example 3

Let's see one more example to test the equality of string present in the list.

1. **import** java.util.ArrayList;
2. **public** **class** EqualsExample3 {
3. **public** **static** **void** main(String[] args) {
4. String str1 = "Mukesh";
5. ArrayList<String> list = **new** ArrayList<>();
6. list.add("Ravi");
7. list.add("Mukesh");
8. list.add("Ramesh");
9. list.add("Ajay");
10. **for** (String str : list) {
11. **if** (str.equals(str1)) {
12. System.out.println("Mukesh is present");
13. }
14. }
15. }
16. }

Mukesh is present

### Java String equalsIgnoreCase()

The **String equalsIgnoreCase()** method compares the two given strings on the basis of content of the string irrespective of case of the string. It is like equals() method but doesn't check case. If any character is not matched, it returns false otherwise it returns true.

#### Java String equalsIgnoreCase() Method Example 2

Let's see an example where we are testing string equality among the strings.

1. **import** java.util.ArrayList;
2. **public** **class** EqualsIgnoreCaseExample2 {
3. **public** **static** **void** main(String[] args) {
4. String str1 = "Mukesh Kumar";
5. ArrayList<String> list = **new** ArrayList<>();
6. list.add("Mohan");
7. list.add("Mukesh");
8. list.add("RAVI");
9. list.add("MuKesH kuMar");
10. list.add("Suresh");
11. **for** (String str : list) {
12. **if** (str.equalsIgnoreCase(str1)) {
13. System.out.println("Mukesh kumar is present");
14. }
15. }
16. }
17. }

Output:

Mukesh kumar is present

### String join()

The **java string join()** method returns a string joined with given delimiter. In string join method, delimiter is copied for each elements.

In case of null element, "null" is added. The join() method is included in java string since JDK 1.8.

There are two types of join() methods in java string.

#### Parameters

**delimiter** : char value to be added with each element

**elements** : char value to be attached with delimiter

#### Returns

joined string with delimiter

#### Throws

**NullPointerException** if element or delimiter is null.

#### Since

**1.8**

#### Java String join() method example

1. **public** **class** StringJoinExample{
2. **public** **static** **void** main(String args[]){
3. String joinString1=String.join("-","welcome","to","javatpoint");
4. System.out.println(joinString1);
5. }}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=StringJoinExample)

welcome-to-javatpoint

#### Java String join() Method Example 2

We can use delimeter to format the string as we did in the below example to show date and time.

1. **public** **class** StringJoinExample2 {
2. **public** **static** **void** main(String[] args) {
3. String date = String.join("/","25","06","2018");
4. System.out.print(date);
5. String time = String.join(":", "12","10","10");
6. System.out.println(" "+time);
7. }
8. }

25/06/2018 12:10:10

### Java String split()

The **java string split()** method splits this string against given regular expression and returns a char array.

## Most Popular Java Programming Interview Questions

### **Most Popular Java Programming Interview Questions**

A list of the most popular Java Programming interview questions and answers are explained below and these questions will help you to clear any Automation Interview successfully.

***Recommended Read => We have covered***[***Core Java Interview Questions***](https://www.softwaretestinghelp.com/core-java-interview-questions/)***in earlier articles here.***

**Q #1) Write a Java Program to reverse a string without using String inbuilt function.**

**Answer:**Here, we are initializing a string variable str and are making use of the string builder class.

The object of the string builder class str2 will be further used to append the value stored in the string variable str.  
Thereafter, we are using the inbuilt function of string builder (reverse()) and storing the new reversed string in str2.

**Finally, we are printing str2.**

|  |  |  |
| --- | --- | --- |
| 1 | public class FinalReverseWithoutUsingStringMethods { | |
| 2 |  |

|  |  |
| --- | --- |
| 3 | public static void main(String[] args) { |
| 4 | // TODO Auto-generated method stub | |

|  |  |
| --- | --- |
| 5 | String str = "Automation"; |
| 6 | StringBuilder str2 = new StringBuilder(); | |

|  |  |
| --- | --- |
| 7 | str2.append(str); |
| 8 | str2 = str2.reverse();     // used string builder to reverse | |

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | System.out.println(str2); | | |
| 10 | | } |

|  |  |  |
| --- | --- | --- |
| 11 |  | |
| 12 | } |

**Output:**

noitamotuA

**Q #2) Write a Java Program to reverse a string without using String inbuilt function reverse().**

**Answer:**

**Method 1:**

There are several ways with which you can reverse your string if you are allowed to use the other string inbuilt functions.

In this method, we are initializing a string variable called str with the value of your given string. Then, we are converting that string into character array with toCharArray() function. Thereafter, we are using for loop to iterate between each character in reverse order and printing each character.

|  |  |  |
| --- | --- | --- |
| 1 | public class FinalReverseWithoutUsingInbuiltFunction { | |
| 2 | public static void main(String[] args) { |

|  |  |
| --- | --- |
| 3 | String str = "Saket Saurav"; |
| 4 | char chars[] = str.toCharArray();  // converted to character array and printed in reverse order | |

|  |  |  |
| --- | --- | --- |
| 5 | for(int i= chars.length-1; i>=0; i--) { | |
| 6 | System.out.print(chars[i]); |

|  |  |  |
| --- | --- | --- |
| 7 | } | |
| 8 | } |

|  |  |
| --- | --- |
| 9 | } |

**Output:**

varuaS tekaS

**Method 2:**

This is another method in which you are declaring your string variable str and then using Scanner class to declare an object with predefined standard input object.

This program will accept the string value through the command line (when executed).

We have used nextLine() which will read the input with the spaces between the words of a string. Thereafter, we have used a split() method to split the string into its substrings(no delimiter given here). Finally, we have printed the string in reverse order using for loop.

|  |  |  |
| --- | --- | --- |
| 1 | import java.util.Scanner; | |
| 2 |  |

|  |  |  |
| --- | --- | --- |
| 3 | public class ReverseSplit { | |
| 4 |  |

|  |  |  |
| --- | --- | --- |
| 5 | public static void main(String[] args) { | |
| 6 | // TODO Auto-generated method stub |

|  |  |
| --- | --- |
| 7 | String str; |
| 8 | Scanner in = new Scanner(System.in); | |

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | System.out.println("Enter your String"); | | |
| 10 | | str = in.nextLine(); |

|  |  |  |
| --- | --- | --- |
| 11 | String[] token = str.split("");    //used split method to print in reverse order | |
| 12 | for(int i=token.length-1; i>=0; i--) |

|  |  |
| --- | --- |
| 13 | { |
| 14 | System.out.print(token[i] + ""); | |

|  |  |  |
| --- | --- | --- |
| 15 | } | |
| 16 |  |

|  |  |  |
| --- | --- | --- |
| 17 | } | |
| 18 |  |

|  |  |
| --- | --- |
| 19 | } |

**Output:**

Enter your String  
Softwaretestinghelp  
plehgnitseterawtfoS

**Method 3:**

This is almost like method 2, but here we did not use the split() method. We have used the scanner class and nextLine() for reading the input string. Then, we have declared an integer length which has the length of the input string.

Thereafter, we have printed the string in the reverse order using for loop. However, we have used charAt(index) method which will return the character at any specific index. After each iteration, the character will be concatenated to reverse the string variable.

Finally, we have printed the reverse string variable.

|  |  |  |
| --- | --- | --- |
| 1 | import java.util.Scanner; | |
| 2 |  |

|  |  |  |
| --- | --- | --- |
| 3 | public class Reverse { | |
| 4 |  |

|  |  |  |
| --- | --- | --- |
| 5 | public static void main(String[] args) { | |
| 6 | // TODO Auto-generated method stub |

|  |  |
| --- | --- |
| 7 | String original, reverse = ""; |
| 8 | System.out.println("Enter the string to be reversed"); | |

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | Scanner in = new Scanner(System.in); | | |
| 10 | | original = in.nextLine(); |

|  |  |
| --- | --- |
| 11 | int length = original.length(); |
| 12 | for(int i=length-1; i>=0; i--) { | |

|  |  |  |
| --- | --- | --- |
| 13 | reverse = reverse + original.charAt(i);   //used inbuilt method charAt() to reverse the string | |
| 14 | } |

|  |  |  |
| --- | --- | --- |
| 15 | System.out.println(reverse); | |
| 16 | } |

|  |  |
| --- | --- |
| 17 |  |
| 18 | } | |

**Output:**

Enter the string to be reversed  
automation testing  
gnitset noitamotua

### Arrays.toString() in Java with Examples

Today we are going to discuss the simplest way to print the array as a string in Java: Arrays.toString() method.

|  |
| --- |
| // Java program to demonstrate working of Arrays.toString()  import java.io.\*;  import java.util.\*;    class GFG {      public static void main(String[] args)      {          // Let us create different types of arrays and          // print their contents using Arrays.toString()          boolean[] boolArr = new boolean[] { true, true, false, true };          byte[] byteArr = new byte[] { 10, 20, 30 };          char[] charArr = new char[] { 'g', 'e', 'e', 'k', 's' };          double[] dblArr = new double[] { 1, 2, 3, 4 };          float[] floatArr = new float[] { 1, 2, 3, 4 };          int[] intArr = new int[] { 1, 2, 3, 4 };          long[] lomgArr = new long[] { 1, 2, 3, 4 };          Object[] objArr = new Object[] { 1, 2, 3, 4 };          short[] shortArr = new short[] { 1, 2, 3, 4 };            System.out.println(Arrays.toString(boolArr));          System.out.println(Arrays.toString(byteArr));          System.out.println(Arrays.toString(charArr));          System.out.println(Arrays.toString(dblArr));          System.out.println(Arrays.toString(floatArr));          System.out.println(Arrays.toString(intArr));          System.out.println(Arrays.toString(lomgArr));          System.out.println(Arrays.toString(objArr));          System.out.println(Arrays.toString(shortArr));      }  } |

Output:

[true, true, false, true]

[10, 20, 30]

[g, e, e, k, s]

[1.0, 2.0, 3.0, 4.0]

[1.0, 2.0, 3.0, 4.0]

[1, 2, 3, 4]

[1, 2, 3, 4]

[1, 2, 3, 4]

[1, 2, 3, 4]

**We can also use Arrays.toString() for objects of user defined class.**  
Since Arrays.toString() is overloaded for array of[Object class](https://www.geeksforgeeks.org/object-class-in-java/) (there exist a method Arrays.toString(Object [])) and Object is ancestor of all classes, we can use call it for an array of any type of object.

|  |
| --- |
| // Java program to demonstrate working of Arrays.toString()  // for user defined objects.  import java.lang.\*;  import java.util.\*;    // Driver class  class Main {      public static void main(String[] args)      {          Student[] arr = { new Student(111, "bbbb", "london"),                            new Student(131, "aaaa", "nyc"),                            new Student(121, "cccc", "jaipur") };            System.out.println(Arrays.toString(arr));      }  }    // A class to represent a student.  class Student {      int rollno;      String name, address;        // Constructor      public Student(int rollno, String name,                     String address)      {          this.rollno = rollno;          this.name = name;          this.address = address;      }        // Used to print student details in main()      @override      public String toString()      {          return this.rollno + " " + this.name + " " + this.address;      }  } |

Output:

[111 bbbb london, 131 aaaa nyc, 121 cccc jaipur]

### Print all the duplicates in the input string

**Write an efficient C program to print all the duplicates and their counts in the input string**

**Algorithm:** Let input string be “geeksforgeeks”  
**1:** Construct character count array from the input string.

count[‘e’] = 4  
count[‘g’] = 2  
count[‘k’] = 2  
……

**2:**Print all the indexes from the constructed array which have value greater than 1.

**Solution**

|  |
| --- |
| // Java program to count all duplicates from string using hashing    public class GFG  {      static final int NO\_OF\_CHARS = 256;        /\* Fills count array with frequency of characters \*/      static void fillCharCounts(String str, int[] count)      {         for (int i = 0; i < str.length();  i++)            count[str.charAt(i)]++;      }        /\* Print duplicates present in the passed string \*/      static void printDups(String str)      {        // Create an array of size 256 and fill count of every character in it        int count[] = new int[NO\_OF\_CHARS];        fillCharCounts(str, count);          for (int i = 0; i < NO\_OF\_CHARS; i++)          if(count[i] > 1)              System.out.printf("%c,  count = %d \n", i,  count[i]);        }        // Driver Method      public static void main(String[] args)      {          String str = "test string";          printDups(str);      }  } |

**Output :**

s, count = 2

t, count = 3

**Time Complexity :**O(n)

### Output of Java Program | Set 11

Predict the output of following Java programs:

**Question 1 :**

|  |
| --- |
| public class Base  {      private int data;        public Base()      {          data = 5;      }        public int getData()      {          return this.data;      }  }    class Derived extends Base  {      private int data;      public Derived()      {          data = 6;      }      private int getData()      {          return data;      }        public static void main(String[] args)      {          Derived myData = new Derived();          System.out.println(myData.getData());      }  } |

a) 6  
b) 5  
c) Compile time error  
d) Run time error

Answer (c)  
**Explanation:** When overriding a method of superclass, [the method declaration in subclass cannot be more restrictive than that declared in the superclass](https://www.geeksforgeeks.org/more-restrictive-access-is-given-to-a-derived-class-method-in-java/).

## Look-and-say sequence

[](https://www.rosettacode.org/wiki/Category:Solutions_by_Programming_Task)

**Look-and-say sequence**  
You are encouraged to [solve this task](https://www.rosettacode.org/wiki/Rosetta_Code:Solve_a_Task) according to the task description, using any language you may know.

The   [Look and say sequence](https://en.wikipedia.org/wiki/Look_and_say_sequence)   is a recursively defined sequence of numbers studied most notably by   [John Conway](https://en.wikipedia.org/wiki/John_Horton_Conway).

**Sequence Definition**

* Take a decimal number
* *Look* at the number, visually grouping consecutive runs of the same digit.
* *Say* the number, from left to right, group by group; as how many of that digit there are - followed by the digit grouped.

This becomes the next number of the sequence.

**An example:**

* Starting with the number 1,   you have *one* 1 which produces 11
* Starting with 11,   you have *two* 1's.   I.E.:   21
* Starting with 21,   you have *one* 2, then *one* 1.   I.E.:   (12)(11) which becomes 1211
* Starting with 1211,   you have *one* 1, *one* 2, then *two* 1's.   I.E.:   (11)(12)(21) which becomes 111221

## Generics in Java

Generics in Java is similar to [templates in C++](http://geeksquiz.com/templates-cpp/). The idea is to allow **type (Integer, String, … etc and user defined types)** to be a parameter to methods, classes and interfaces. For example, classes like HashSet, ArrayList, HashMap, etc use generics very well. We can use them for any type.

<https://www.geeksforgeeks.org/generics-in-java/>

**Generic Class**

Like C++, we use <> to specify parameter types in generic class creation. To create objects of generic class, we use following syntax.

// To create an instance of generic class

**BaseType <Type> obj = new BaseType <Type>()**

**Note:** In Parameter type we can not use primitives like

'int','char' or 'double'.

|  |
| --- |
| // A Simple Java program to show working of user defined  // Generic classes    // We use < > to specify Parameter type  class Test<T>  {      // An object of type T is declared      T obj;      Test(T obj) {  this.obj = obj;  }  // constructor      public T getObject()  { return this.obj; }  }    // Driver class to test above  class Main  {      public static void main (String[] args)      {          // instance of Integer type          Test <Integer> iObj = new Test<Integer>(15);          System.out.println(iObj.getObject());            // instance of String type          Test <String> sObj =                            new Test<String>("GeeksForGeeks");          System.out.println(sObj.getObject());      }  } |

Output:

15

GeeksForGeeks

We can also pass multiple Type parameters in Generic classes.

|  |
| --- |
| // A Simple Java program to show multiple  // type parameters in Java Generics    // We use < > to specify Parameter type  class Test<T, U>  {      T obj1;  // An object of type T      U obj2;  // An object of type U        // constructor      Test(T obj1, U obj2)      {          this.obj1 = obj1;          this.obj2 = obj2;      }        // To print objects of T and U      public void print()      {          System.out.println(obj1);          System.out.println(obj2);      }  }    // Driver class to test above  class Main  {      public static void main (String[] args)      {          Test <String, Integer> obj =              new Test<String, Integer>("GfG", 15);            obj.print();      }  } |

Output:

GfG

15

**Generic Functions:**

We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to generic method, the compiler handles each method.

|  |
| --- |
| // A Simple Java program to show working of user defined  // Generic functions    class Test  {      // A Generic method example      static <T> void genericDisplay (T element)      {          System.out.println(element.getClass().getName() +                             " = " + element);      }        // Driver method      public static void main(String[] args)      {           // Calling generic method with Integer argument          genericDisplay(11);            // Calling generic method with String argument          genericDisplay("GeeksForGeeks");            // Calling generic method with double argument          genericDisplay(1.0);      }  } |

Output :

java.lang.Integer = 11

java.lang.String = GeeksForGeeks

java.lang.Double = 1.0

**Advantages of Generics:**

Programs that uses Generics has got many benefits over non-generic code.

1. Code Reuse: We can write a method/class/interface once and use for any type we want.

.

1. Type Safety : Generics make errors to **appear compile time** than at run time (It’s always better to know problems in your code at compile time rather than making your code fail at run time). **Suppose you want to create an ArrayList that store name of students and if by mistake programmer adds an integer object instead of string, compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.**

|  |
| --- |
| // A Simple Java program to demonstrate that NOT using  // generics can cause run time exceptions  import java.util.\*;    class Test  {      public static void main(String[] args)      {          // Creating a an ArrayList without any type specified          ArrayList al = new ArrayList();            al.add("Sachin");          al.add("Rahul");          al.add(10); // Compiler allows this            String s1 = (String)al.get(0);          String s2 = (String)al.get(1);            // Causes Runtime Exception          String s3 = (String)al.get(2);      }  } |

Output :

Exception in thread "main" java.lang.ClassCastException:

java.lang.Integer cannot be cast to java.lang.String

at Test.main(Test.java:19)

**How generics solve this problem?**  
At the time of defining ArrayList, we can specify that this list can take only String objects.

|  |
| --- |
| // Using generics converts run time exceptions into  // compile time exception.  import java.util.\*;    class Test  {      public static void main(String[] args)      {          // Creating a an ArrayList with String specified          ArrayList <String> al = new ArrayList<String> ();            al.add("Sachin");          al.add("Rahul");            // Now Compiler doesn't allow this          al.add(10);            String s1 = (String)al.get(0);          String s2 = (String)al.get(1);          String s3 = (String)al.get(2);      }  } |

Output:

15: error: no suitable method found for add(int)

al.add(10);

^

.

1. Individual Type Casting is not needed: If we do not use generics, then, in the above example every-time we retrieve data from ArrayList, we have to typecast it. Typecasting at every retrieval operation is a big headache. If we already know that our list only holds string data then we need not to typecast it every time.

|  |
| --- |
| // We don't need to typecast individual members of ArrayList  import java.util.\*;    class Test  {      public static void main(String[] args)      {          // Creating a an ArrayList with String specified          ArrayList <String> al = new ArrayList<String> ();            al.add("Sachin");          al.add("Rahul");            // Typecasting is not needed          String s1 = al.get(0);          String s2 = al.get(1);      }  } |

.

1. Implementing generic algorithms: By using generics, we can implement algorithms that work on different types of objects and at the same they are type safe too.

## What is the benefit of Generics in Collections Framework?

Java 1.5 came with Generics and all collection interfaces and implementations use it heavily. Generics allow us to provide the type of Object that a collection can contain, so if you try to add any element of other type it throws compile time error.  
This avoids ClassCastException at Runtime because you will get the error at compilation. Also Generics make code clean since we don’t need to use casting and instanceof operator.

## ----------------------------------------------------------------------------------                                   *String*                *StringBuffer*        *StringBuilder* ----------------------------------------------------------------------------------                 Storage Area | Constant String Pool           Heap                       Heap Modifiable     |  No (immutable)            Yes( mutable )          Yes( mutable ) Thread Safe   |           Yes                                  Yes                              No  Performance |         Slow                       Slow but faster             Fast                                                                    than String

## File-Handling

### Java.io.File Class in Java

The File class is Java’s representation of a file or directory path name. Because file and directory names have different formats on different platforms, a simple string is not adequate to name them. The File class contains several methods for working with the path name, deleting and renaming files, creating new directories, listing the contents of a directory, and determining several common attributes of files and directories.

* It is an abstract representation of file and directory pathnames.
* A pathname, whether abstract or in string form can be either absolute or relative. The parent of an abstract pathname may be obtained by invoking the getParent() method of this class.
* First of all, we should create the File class object by passing the filename or directory name to it. A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing. These restrictions are collectively known as access permissions.
* Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change.

**How to create a File Object?**  
A File object is created by passing in a String that represents the name of a file, or a String or another File object. For example,

**File a = new File("/usr/local/bin/geeks");**

defines an abstract file name for the geeks file in directory /usr/local/bin. This is an absolute abstract file name.

**Constructors**

* **File(File parent, String child) :**Creates a new File instance from a parent abstract pathname and a child pathname string.
* **File(String pathname) :**Creates a new File instance by converting the given pathname string into an abstract pathname.
* **File(String parent, String child) :**Creates a new File instance from a parent pathname string and a child pathname string.
* **File(URI uri) :**Creates a new File instance by converting the given file: URI into an abstract pathname.

**Methods**

1. **boolean canExecute() :**Tests whether the application can execute the file denoted by this abstract pathname.
2. **boolean canRead()**: Tests whether the application can read the file denoted by this abstract pathname.
3. **boolean canWrite() :**Tests whether the application can modify the file denoted by this abstract pathname.
4. **int compareTo(File pathname) :**Compares two abstract pathnames lexicographically.
5. **boolean createNewFile() :**Atomically creates a new, empty file named by this abstract pathname .
6. **static File createTempFile(String prefix, String suffix) :**Creates an empty file in the default temporary-file directory.
7. **boolean delete() :**Deletes the file or directory denoted by this abstract pathname.
8. **boolean equals(Object obj) :**Tests this abstract pathname for equality with the given object.
9. **boolean exists()**: Tests whether the file or directory denoted by this abstract pathname exists.
10. **String getAbsolutePath() :**Returns the absolute pathname string of this abstract pathname.
11. **long getFreeSpace() :**Returns the number of unallocated bytes in the partition .
12. **String getName() :**Returns the name of the file or directory denoted by this abstract pathname.
13. **String getParent() :**Returns the pathname string of this abstract pathname’s parent.
14. **File getParentFile() :**Returns the abstract pathname of this abstract pathname’s parent.
15. **String getPath() :**Converts this abstract pathname into a pathname string.
16. **boolean isDirectory() :**Tests whether the file denoted by this pathname is a directory.
17. **boolean isFile() :**Tests whether the file denoted by this abstract pathname is a normal file.
18. **boolean isHidden() :**Tests whether the file named by this abstract pathname is a hidden file.
19. **long length() :**Returns the length of the file denoted by this abstract pathname.
20. **String[] list() :**Returns an array of strings naming the files and directories in the directory .
21. **File[] listFiles() :**Returns an array of abstract pathnames denoting the files in the directory.
22. **boolean mkdir() :**Creates the directory named by this abstract pathname.
23. **boolean renameTo(File dest) :**Renames the file denoted by this abstract pathname.
24. **boolean setExecutable(boolean executable) :**A convenience method to set the owner’s execute permission.
25. **boolean setReadable(boolean readable) :**A convenience method to set the owner’s read permission.
26. **boolean setReadable(boolean readable, boolean ownerOnly) :**Sets the owner’s or everybody’s read permission.
27. **boolean setReadOnly() :**Marks the file or directory named so that only read operations are allowed.
28. **boolean setWritable(boolean writable)**: A convenience method to set the owner’s write permission.
29. **String toString() :**Returns the pathname string of this abstract pathname.
30. **URI toURI() :**Constructs a file URI that represents this abstract pathname.

**Implementation**

**Program 1:**Program to check if a file or directory physically exist or not.

|  |
| --- |
| // In this program, we accepts a file or directory name from  // command line arguments. Then the program will check if  // that file or directory physically exist or not and  // it displays the property of that file or directory.  \*import java.io.File;    // Displaying file property  class fileProperty  {      public static void main(String[] args) {          //accept file name or directory name through command line args          String fname =args[0];            //pass the filename or directory name to File object          File f = new File(fname);            //apply File class methods on File object          System.out.println("File name :"+f.getName());          System.out.println("Path: "+f.getPath());          System.out.println("Absolute path:" +f.getAbsolutePath());          System.out.println("Parent:"+f.getParent());          System.out.println("Exists :"+f.exists());          if(f.exists())          {              System.out.println("Is writeable**:"+f.canWrite());**              System.out.println("Is readable"+f.canRead());              System.out.println("Is a directory:"+f.isDirectory());              System.out.println("File Size in bytes "+f.length());          }      }  } |

Output:

File name :file.txt

Path: file.txt

Absolute path:C:\Users\akki\IdeaProjects\codewriting\src\file.txt

Parent:null

Exists :true

Is writeable:true

Is readabletrue

Is a directory:false

File Size in bytes 20

**Program 2:**Program to display all the contents of a directory

Here we will accept a directory name from the keyboard and then display all the contents of the directory .For this purpose, list() method can be used as:

String arr[]=f.list();

In the preceding statement , the list() method causes all the directory entries copied into the array arr[]. Then pass these array elements arr[i] to File object and test them to know if they represent a file or directory .

|  |
| --- |
| import java.io.BufferedReader;  import java.io.File;  import java.io.IOException;  import java.io.InputStreamReader;    //Displaying the contents of a directory  class Contents  {      public static void main(String[] args) throws IOException {          //enter the path and dirname from keyboard          BufferedReader br = new BufferedReader(new InputStreamReader(System.in));            System.out.println("Enter dirpath:");          String dirpath=br.readLine();          System.out.println("Enter the dirname");          String dname=br.readLine();            //create File object with dirpath and dname          File f = new File(dirpath, dname);            //if directory exists,then          if(f.exists())          {              //get the contents into arr[]              //now arr[i] represent either a File or Directory              String arr[]=f.list();                //find no. of entries in the directory              int n=arr.length;                //displaying the entries              for (int i = 0; i < n ; i++) {                  System.out.println(arr[i]);                  //create File object with the entry and test                  //if it is a file or directory                  File f1=new File(arr[i]);                  if(f1.isFile())                      System.out.println(": is a file");                  if(f1.isDirectory())                      System.out.println(": is a directory");              }              System.out.println("No of entries in this directory "+n);          }          else              System.out.println("Directory not found");      }  } |

Output:

Enter dirpath:

C:\Users\akki\IdeaProjects\

Enter the dirname

codewriting

.idea

: is a directory

an1.txt

: is a file

codewriting.iml

: is a file

file.txt

: is a file

out

: is a directory

src

: is a directory

text

: is a file

No of entries in this directory 7

### File handling in Java using FileWriter and FileReader

**Java FileWriter and FileReader classes** are used to write and read data from text files (they are [Character Stream](https://www.geeksforgeeks.org/character-stream-vs-byte-stream-java/) classes). It is recommended **not** to use the FileInputStream and FileOutputStream classes if you have to read and write any textual information as these are Byte stream classes.

**FileWriter**

FileWriter is useful to create a file writing characters into it.

* This class inherits from the OutputStream class.
* The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an OutputStreamWriter on a FileOutputStream.
* FileWriter is meant for **writing streams of characters.** For writing streams of raw bytes, consider using a FileOutputStream.
* **FileWriter creates the output file , if it is not present already.**

**Constructors:**

* **FileWriter(File file) –** Constructs a FileWriter object given a File object.
* **FileWriter (File file, boolean append) –** constructs a FileWriter object given a File object.
* **FileWriter (FileDescriptor fd) –** constructs a FileWriter object associated with a file descriptor.
* **FileWriter (String fileName) –** constructs a FileWriter object given a file name.
* **FileWriter (String fileName, Boolean append) –** Constructs a FileWriter object given a file name with a Boolean indicating whether or not to append the data written.

**Methods:**

* **public void write (int c) throws IOException –** Writes a single character.
* **public void write (char [] stir) throws IOException –** Writes an array of characters.
* **public void write(String str)throws IOException –** Writes a string.
* **public void write(String str,int off,int len)throws IOException –** Writes a portion of a string. Here off is offset from which to start writing characters and len is number of character to write.
* **public void flush() throws IOException** flushes the stream
* **public void close() throws IOException** flushes the stream first and then closes the writer.

Reading and writing take place character by character, which increases the number of I/O operations and effects performance of the system.**BufferedWriter** can be used along with FileWriter to improve speed of execution.

Following program depicts how to create a text file using FileWriter

|  |
| --- |
| // Creating a text File using FileWriter  import java.io.FileWriter;  import java.io.IOException;  class CreateFile  {      public static void main(String[] args) throws IOException      {          // Accept a string          String str = "File Handling in Java using "+                  " FileWriter and FileReader";            // attach a file to FileWriter          FileWriter fw=new FileWriter("output.txt");            // read character wise from string and write          // into FileWriter          for (int i = 0; i < str.length(); i++)              fw.write(str.charAt(i));            System.out.println("Writing successful");          //close the file          fw.close();      }  } |

**FileReader**

FileReader is useful to read data in the form of characters from a ‘text’ file.

* This class inherit from the **InputStreamReader** Class.
* The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate. To specify these values yourself, construct an InputStreamReader on a FileInputStream.
* FileReader is meant for reading streams of characters. For reading streams of raw bytes, consider using a FileInputStream.

**Constructors:**

* **FileReader(File file) –**Creates a FileReader , given the File to read from
* **FileReader(FileDescripter fd) –** Creates a new FileReader , given the FileDescripter to read from
* **FileReader(String fileName) –**Creates a new FileReader , given the name of the file to read from

**Methods:**

* **public int read () throws IOException –** Reads a single character. This method will block until a character is available, an I/O error occurs, or the end of the stream is reached.
* **public int read(char[] cbuff) throws IOException –** Reads characters into an array. This method will block until some input is available, an I/O error occurs, or the end of the stream is reached.
* **public abstract int read(char[] buff, int off, int len) throws IOException –**Reads characters into a portion of an array. This method will block until some input is available, an I/O error occurs, or the end of the stream is reached.  
  Parameters:  
  cbuf – Destination buffer  
  off – Offset at which to start storing characters  
  len – Maximum number of characters to read
* **public void close() throws IOException** closes the reader.
* **public long skip(long n) throws IOException –**Skips characters. This method will block until some characters are available, an I/O error occurs, or the end of the stream is reached.  
  Parameters:  
  n – The number of characters to skip

Following program depicts how to read from the ‘text’ file using FileReader

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| // Reading data from a file using FileReader  import java.io.FileNotFoundException;  import java.io.FileReader;  import java.io.IOException;  class ReadFile  {      public static void main(String[] args) throws IOException      {          // variable declaration          int ch;            // check if File exists or not          FileReader fr=null;          try          {              fr = new FileReader("text");          }          catch (FileNotFoundException fe)          {              System.out.println("File not found");          }            // read from FileReader till the end of file          while ((ch=fr.read())!=-1)              System.out.print((char)ch);            // close the file          fr.close();      }  }  Java.io.BufferedWriter class methods in Java  [io.BufferedWriter class methods](https://media.geeksforgeeks.org/wp-content/uploads/io.BufferedWriter-class-methods.jpg)  Bufferreader class writes text to character-output stream, buffering characters.Thus, providing efficient writing of single array, character and strings. A buffer size needs to be specified, if not it takes Default value. An output is immediately set to the underlying character or byte stream by the Writer.  **Class Declaration**    public class BufferedWriter  extends Writer  **Constructors**   * **BufferedWriter(Writer out)**: Creates a buffered character-output stream that uses a default-sized output buffer. * **BufferedWriter(Writer out, int size):** Creates a new buffered character-output stream that uses an output buffer of the given size.   **Methods:**   * **write() : java.io.BufferedWriter.write(int arg)** writes a single character that is specified by an integer argument. **Syntax :** * public void write(int arg) * **Parameters :** * arg : integer that specifies the character to write * **Return :** * Doesn't return any value.   **Implementation :**  filter\_none  edit  play\_arrow  brightness\_4   |  | | --- | | //Java program illustrating use of write(int arg) method    import java.io.\*;  public class NewClass  {      public static void main(String[] args)      {          //initializing FileWriter          FileWriter geek\_file;          try          {              geek\_file = new FileWriter("ABC.txt");                // Initialing BufferedWriter              BufferedWriter geekwrite = new BufferedWriter(geek\_file);              System.out.println("Buffered Writer start writing :)");                // Use of write() method to write the value in 'ABC' file              // Printing E              geekwrite.write(69);                // Printing 1              geekwrite.write(49);                // Closing BufferWriter to end operation              geekwrite.close();              System.out.println("Written successfully");          }          catch (IOException excpt)          {              excpt.printStackTrace();          }        }  } |   **Note :** In the given output, you can’t see it’s action on file. Run this code on any compiler in your device. It creates a new file ‘ABC’ and write “E 1 ” in it.  Output :  Buffered Writer start writing :)  Written successfully   * **write() : java.io.BufferedWriter.write(String arg, int offset, int length)** writes String in the file according to its arguments as mentioned in the Java Code. **Syntax :** * public void write(String arg, int offset, int length) * **Parameters :** * arg : String to be written * offset : From where to start reading the string * length : No. of characters of the string to write * **Return :** * Doesn't return any value.   **Implementation :**  filter\_none  edit  play\_arrow  brightness\_4   |  | | --- | | //Java program illustrating use of write(String arg, int offset, int length) method    import java.io.\*;  public class NewClass  {      public static void main(String[] args)      {          //Initializing a FileWriter          FileWriter geek\_file;          try          {              geek\_file = new FileWriter("ABC.txt");                // Initializing a BufferedWriter              BufferedWriter geekwrite = new BufferedWriter(geek\_file);              System.out.println("Buffered Writer start writing :)");              String arg = "Hello Geeks";              int offset = 6;              geekwrite.write(arg,offset,arg.length()-offset);                // Closing Buffer              geekwrite.close();              System.out.println("Written successfully");          }          catch (IOException except)          {              except.printStackTrace();          }          }  } |   **Note :**In the given output, you can’t see it’s action on file. Run this code on any compiler in your device. It creates a new file ‘ABC’ and write “Geeks” in it.Here,  arg = Hello Geeks  offset = 6  length = arg.length So, when we minus offset : 6, it will write 'Geeks' only in the file.  Output:  Buffered Writer start writing :)  Written successfully   * **newLine() : java.io.BufferedWriter.newLine()** breaks/separates line. **Syntax :** * public void newLine() * **Return :** * Doesn't return any value.   **Implementation :**  filter\_none  edit  play\_arrow  brightness\_4   |  | | --- | | //Java program explaining use of newLine() method    import java.io.\*;  public class NewClass  {      public static void main(String[] args)      {          //initializing FileWriter          FileWriter geek\_file;          try          {              geek\_file = new FileWriter("ABC.txt");                // Initialing BufferedWriter              BufferedWriter geekwrite = new BufferedWriter(geek\_file);              System.out.println("Buffered Writer start writing :)");                // Use of write() method to write the value in 'ABC' file              // Printing "GEEKS"              geekwrite.write("GEEKS");                // For next line              geekwrite.newLine();                // Printing "FOR"              geekwrite.write("FOR");                 // For next line              geekwrite.newLine();                // Printing "GEEKS"              geekwrite.write("FOR");                // Closing BufferWriter to end operation              geekwrite.close();              System.out.println("Written successfully");          }          catch (IOException excpt)          {              excpt.printStackTrace();          }        }  } |   **Note :**In the given output, you can’t see it’s action on file. Run this code on any compiler in your device. It creates a new file ‘ABC’ and write write | GEEKS | | FOR | | GEEKS | Here, newLine() method breaks line after GEEKS and FOR is written in next line Output :  Buffered Writer start writing :)  Written successfully   * **flush() : java.io.BufferedWriter.flush()** flushes character from write buffer. **Syntax :** * public void flush() * **Return :** * Doesn't return any value. * **close() : java.io.BufferedWriter.close()** flushes character from write buffer and then close it. **Syntax :** * public void close() * **Return :** * Doesn't return any value.   **Implementation of flush(), close() method :**  filter\_none  edit  play\_arrow  brightness\_4   |  | | --- | | //Java program illustrating use of flush(), close() method    import java.io.\*; //BufferedWriter, FileWriter, IOException  public class NewClass  {      public static void main(String[] args)      {          FileWriter geek\_file; //initializing FileWriter          try          {              geek\_file = new FileWriter("ABC.txt");              // Initialing BufferedWriter              BufferedWriter geekwrite = new BufferedWriter(geek\_file);              System.out.println("Buffered Writer start writing :)");              // Use of write() method to write the value in 'ABC' file                geekwrite.write(69); // Printing E              geekwrite.newLine(); // For next line              geekwrite.write(49); // Printing 1                // flush() method : flushing the stream              geekwrite.flush();              // close() method : closing BufferWriter to end operation              geekwrite.close();              System.out.println("Written successfully");          }          catch (IOException excpt)          {              excpt.printStackTrace();          }        }  } |   **Note :** You can’t see it’s action on file. Run this code on any compiler in your device.It creates a new file ‘ABC’ and write | E | | 1 | in it.Here, flush() method flushes the stream and close() method closes the writer.  Output :  Buffered Writer start writing :)  Written successfully Java Write to File [java write to file, write file in java](https://cdn.journaldev.com/wp-content/uploads/2012/12/java-write-to-file.jpg)  Let’s have a brief look at four options we have for java write to file operation.   1. **FileWriter**: FileWriter is the simplest way to write a file in Java. It provides overloaded write method to write int, byte array, and String to the File. You can also write part of the String or byte array using FileWriter. FileWriter writes directly into Files and should be used only when the number of writes is less. 2. **BufferedWriter**: BufferedWriter is almost similar to FileWriter but it uses internal buffer to write data into File. So if the number of write operations is more, the actual IO operations are less and performance is better. You should use BufferedWriter when the number of write operations is more. 3. **FileOutputStream**: FileWriter and BufferedWriter are meant to write text to the file but when you need raw stream data to be written into file, you should use FileOutputStream to write file in java. 4. **Files**: Java 7 introduced Files utility class and we can write a file using its write function. Internally it’s using OutputStream to write byte array into file.  Java Write to File Example Here is the example showing how we can write a file in java using FileWriter, BufferedWriter, FileOutputStream, and Files in java.  WriteFile.java  package com.journaldev.files;  import java.io.BufferedWriter;  import java.io.File;  import java.io.FileOutputStream;  import java.io.FileWriter;  import java.io.IOException;  import java.io.OutputStream;  import java.nio.file.Files;  import java.nio.file.Paths;  public class WriteFile {  /\*\*  \* This class shows how to write file in java  \* @param args  \* @throws IOException  \*/  public static void main(String[] args) {  String data = "I will write this String to File in Java";  int noOfLines = 10000;  writeUsingFileWriter(data);    writeUsingBufferedWriter(data, noOfLines);    writeUsingFiles(data);    writeUsingOutputStream(data);  System.out.println("DONE");  }  /\*\*  \* Use Streams when you are dealing with raw data  \* @param data  \*/  private static void writeUsingOutputStream(String data) {  OutputStream os = null;  try {  os = new FileOutputStream(new File("/Users/pankaj/os.txt"));  os.write(data.getBytes(), 0, data.length());  } catch (IOException e) {  e.printStackTrace();  }finally{  try {  os.close();  } catch (IOException e) {  e.printStackTrace();  }  }  }    /\*\*  \* Use Files class from Java 1.7 to write files, internally uses OutputStream  \* @param data  \*/  private static void writeUsingFiles(String data) {  try {  Files.write(Paths.get("/Users/pankaj/files.txt"), data.getBytes());  } catch (IOException e) {  e.printStackTrace();  }  }  /\*\*  \* Use BufferedWriter when number of write operations are more  \* It uses internal buffer to reduce real IO operations and saves time  \* @param data  \* @param noOfLines  \*/  private static void writeUsingBufferedWriter(String data, int noOfLines) {  File file = new File("/Users/pankaj/BufferedWriter.txt");  FileWriter fr = null;  BufferedWriter br = null;  String dataWithNewLine=data+System.getProperty("line.separator");  try{  fr = new FileWriter(file);  br = new BufferedWriter(fr);  for(int i = noOfLines; i>0; i--){  br.write(dataWithNewLine);  }  } catch (IOException e) {  e.printStackTrace();  }finally{  try {  br.close();  fr.close();  } catch (IOException e) {  e.printStackTrace();  }  }  }  /\*\*  \* Use FileWriter when number of write operations are less  \* @param data  \*/  private static void writeUsingFileWriter(String data) {  File file = new File("/Users/pankaj/FileWriter.txt");  FileWriter fr = null;  try {  fr = new FileWriter(file);  fr.write(data);  } catch (IOException e) {  e.printStackTrace();  }finally{  //close resources  try {  fr.close();  } catch (IOException e) {  e.printStackTrace();  }  }  }  } |

## Exception Handling- Output of Java program | Set 12()

**Prerequisites :**[Exception handling](https://www.geeksforgeeks.org/java/#Exception%20Handling) , [control flow in try-catch-finally](https://www.geeksforgeeks.org/flow-control-in-try-catch-finally-in-java/)  
**1) What is the output of the following program?**

|  |
| --- |
| public class Test  {      public static void main(String[] args)      {          try          {              System.out.printf("1");              int sum = 9 / 0;              System.out.printf("2");          }          catch(ArithmeticException e)          {              System.out.printf("3");          }          catch(Exception e)          {              System.out.printf("4");          }          finally          {              System.out.printf("5");          }      }  } |

a) 1325  
b) 1345  
c) 1342  
d) 135

**Ans.**(d)  
**Explanation: Once an exception occurs in try block, the execution passes to corresponding catch statement and doesn’t return back to try block. Only one of the catch blocks are executed at a time. finally block is always executed whether or not the exception occurred.**

### 2) What is the output of the following program?

|  |
| --- |
| public class Test  {      private void m1()      {          m2();          System.out.printf("1");      }      private void m2()      {          m3();          System.out.printf("2");      }      private void m3()      {          System.out.printf("3");          try          {              int sum = 4/0;              System.out.printf("4");          }          catch(ArithmeticException e)          {              System.out.printf("5");          }              System.out.printf("7");      }      public static void main(String[] args)      {          Test obj = new Test();          obj.m1();      }  } |

* 1. 35721  
     b) 354721  
     c) 3521  
     d) 35  
     **Ans.**(a)  
     **Explanation:**If an exception is handled in the catch statement, the program continues with its normal execution, after executing the catch statement corresponding to that exception. Also, when an exception occurs in the try block, the rest of the program in the try block is not executed.

### 3) What is the output of the following program?

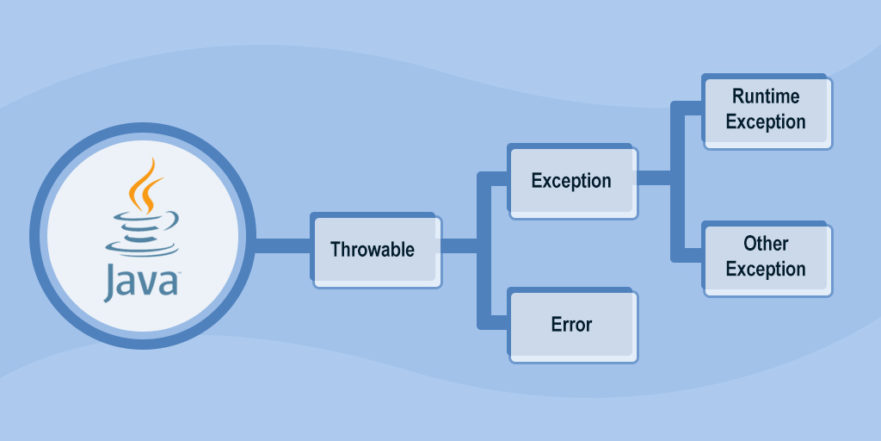
|  |
| --- |
| public class Test  {      public static void main(String[] args)      {          try          {              System.out.printf("1");              int data = 5 / 0;          }          catch(ArithmeticException e)          {              System.out.printf("2");              System.exit(0);          }          finally          {              System.out.printf("3");          }          System.out.printf("4");      }  } |

* a) 12  
  b) 1234  
  c) 124  
  d) 123
* **Ans.**(a)  
  **Explanation:**The only case when the code inside finally block is not executed is when System.exit(0) is called explicitly in the program. Then exit statement is called and the program is terminated without executing any further.

### 5) What is the output of the following program?

|  |
| --- |
| import java.io.EOFException;  import java.io.IOException;    public class Test  {      public static void main(String[] args)      {          try          {              System.out.printf("1");              int value = 10 / 0;              throw new IOException();          }          catch(EOFException e)          {              System.out.printf("2");          }          catch(ArithmeticException e)          {              System.out.printf("3");          }          catch(NullPointerException e)          {              System.out.printf("4");          }          catch(IOException e)          {              System.out.printf("5");          }          catch(Exception e)          {              System.out.printf("6");          }      }  } |

* a) 1346  
  b) 136726  
  c) 136  
  d) 13  
  **Ans.**(d)  
  **Explanation:**In multi-catch statements, the exceptions must be listed from more specific to more general. Only one catch statement which is most specific to the occurred exception is executed.



## Types of Exceptions in Java

SAGAR ARORASEPTEMBER 26, 2018[DEVELOPER TIPS, TRICKS & RESOURCES](https://stackify.com/developers/)

**Java**is an object-oriented programming language. It provides support for various mechanisms such as [**exception handling**](https://stackify.com/specify-handle-exceptions-java/). This feature of Java enables developers to manage the runtime errors caused by the exceptions.

In this article, you will learn about **exceptions in Java**. You will also learn about **different types of exceptions in Java**.

Exceptions are the unwanted errors or bugs or events that restrict the normal execution of a program. Each time an exception occurs, program execution gets disrupted. An error message is displayed on the screen.

There are several reasons behind the occurrence of exceptions. These are some conditions where an exception occurs:

* Whenever a **user provides invalid** data.
* The file requested **to be accessed does not exist in the system**.
* When the **Java Virtual Machine** (JVM) runs out of memory.
* **Network drops** in the middle of communication.

Now let us explore different types of exceptions in Java.

The parent class of all the exception classes is the **java.lang.Exception** class. Figure 1 illustrates the different types of Java exceptions.

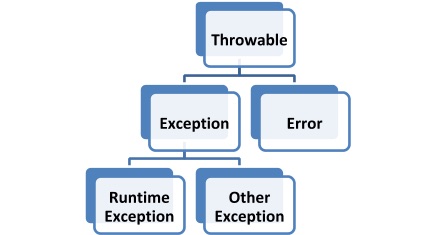
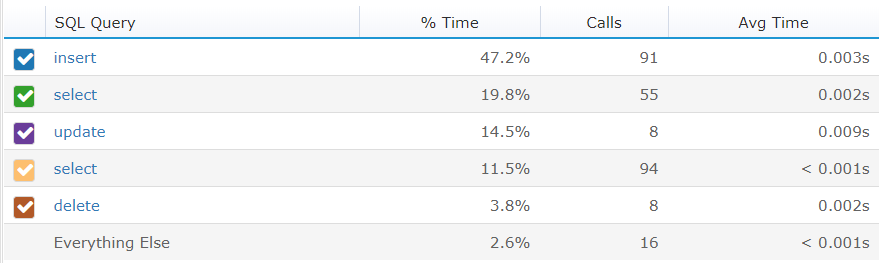


FIGURE 1: TYPES OF EXCEPTIONS IN JAVA



If we talk about the **Exception** class, it is a subclass of the built-in **Throwable** class. There is another subclass which is derived from the Throwable class i.e. **Error** as illustrated in Figure 1. The error can be defined as an abnormal condition that indicates something has gone wrong with the execution of the program. These are not handled by Java programs.

There are some important methods available in the **Throwable** class which are as follows:

* **public String getMessage()**– Provides information about the exception that has occurred through a message, which is initialized in the Throwable constructor.
* **public Throwable getCause()**– Provides root cause of the exception as represented by a Throwable object.
* **public void printStackTrace()**– Used to display the output of toString() along with the stack trace to System.err (error output stream).
* **public StackTraceElement [] getStackTrace()**– Returns an array with each element present on the stack trace. The index 0 element will symbolize the top of the call stack, and the last element of array will identify the bottom of the call stack.

There are mainly two types of exceptions in Java as follows:

* Checked exception
* Unchecked exception

### **Checked exception**

Checked exceptions are also known as compile-time exceptions as these exceptions are checked by the compiler during the compilation process to confirm whether the exception is handled by the programmer or not. If not, then the system displays a compilation error. For example, **SQLException**, **IOException**, **InvocationTargetException,** and **ClassNotFoundException**.

To illustrate the concept of checked exception, let us consider the following code snippet:

import java.io.\*;

class demo1 {

public static void main(String args[]) {

FileInputStream input1 = null;

/\* FileInputStream(File filename) is a constructor that will throw

\* FileNotFoundException (a checked exception)

\*/

input1 = new FileInputStream("D:/file.txt");

int m;

// The read() of FileInputStream will also throw a checked exception

while(( m = input1.read() ) != -1) {

System.out.print((char)m);

}

// The close() will close the file input stream, and it will also throw a exception

input1.close();

}

}

Output:

Exception in thread "main" java.lang.Error: Unresolved compilation problems:

Unhandled exception type FileNotFoundException

Unhandled exception type IOException

Unhandled exception type IOException

#### **throw keyword**

It is clearly displayed in the output that the program throws exceptions during the compilation process. There are two methods of resolving such issues. You can declare the exception with the help of the throw keyword.

import java.io.\*;

class demo1 {

public static void main(String args[]) **throws IOException** {

FileInputStream input1 = null;

input1 = new FileInputStream("D:/file.txt");

int m;

while ((m = input1.read()) != -1) {

System.out.print((char)m);

}

input1.close();

}

}

Output: The file will be displayed on the screen.

#### **try-catch block**

Apart from the above-mentioned method, there is another way to resolve exceptions. You can manage them with the help of **try-catch blocks**.

import java.io.\*;

class demo1 {

public static void main(String args[]) {

FileInputStream input1 = null;

try {

input1 = new FileInputStream("D:/file.txt");

} catch(FileNotFoundException input2) {

system.out.println("The file does not " + "exist at the location");

}

int m;

try {

while((m = input1.read()) != -1) {

System.out.print((char)m);

}

input1.close();

} catch(IOException input3) {

system.out.println("I/O error occurred: "+ input3);

}

}

}

Output: The code will run smoothly and the file will be displayed.

Now, let us learn about other checked exceptions. Some of them are:

#### **SQLException**

This type of exception occurs while executing queries on a database related to the SQL syntax. For example, consider the following code snippet:

public void setClientInfo(String sname, String svalue) throws **SQLClientInfoException** {

try {

checkClosed();

((**java.sql.Connection**) this.mc).setClientInfo(sname, svalue);

} catch (**SQLException** sqlEx) {

try {

checkAndFireConnectionError(sqlEx);

} catch (SQLException sqlEx2) {

SQLClientInfoException client\_Ex = new SQLClientInfoException();

client\_Ex.initCause(sqlEx2);

throw client\_Ex;

}

}

}

Output: This code will generate a SQLException.

#### **IOException**

This type of exception occurs while using file I/O stream operations. For example, consider the following code snippet:

import java.io.\*;

public class sample\_IOException {

private static String filepath = "D:\User\guest\Desktop\File2.txt";

public static void main(String[] args) {

BufferedReader br1 = null;

String curline;

try {

br1 = new BufferedReader(new FileReader(filepath));

while ((curline = br1.readLine()) != null) {

System.out.println(curline);

}

} catch (IOException e) {

System.err.println("IOException found :" + e.getMessage());

} finally {

try {

if (br1 != null)

br1.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

Output: This code will generate an IOException.

#### **ClassNotFoundException**

This type of exception is thrown when the JVM is not able to find the required class. It may be due to a command-line error, a classpath issue, or a missing .class file. For example, consider the following code snippet:

public class sample\_ClassNotFoundException {

private static final String CLASS\_TO\_LOAD = "main.java.Utils";

public static void main(String[] args) {

try {

Class loadedClass = Class.forName(CLASS\_TO\_LOAD);

System.out.println("Class " + loadedClass + " found!");

} catch (ClassNotFoundException ex) {

System.err.println("ClassNotFoundException was found: " + ex.getMessage());

ex.printStackTrace();

}

}

}

Output: This code will generate a ClassNotFoundException.

#### **InvocationTargetException**

This type of exception wraps an exception thrown by an invoked method or a constructor. The thrown exception can be accessed with the help of the getTargetException method. For example, consider the following code snippet:

package main.samplejava;

import java.lang.reflect.InvocationTargetException;

import java.lang.reflect.Method;

public class Example {

@SuppressWarnings("unused")

private int test\_sample(String s1) {

if (s1.length() == 0)

throw new IllegalArgumentException("The string should have at least one character!");

System.out.println("Inside test\_sample: argument's value equals to: "" + s1 + """);

return 0;

}

public static void main(String... args) {

try {

Class<?> c1 = Class.forName("main.samplejava. Example");

Object t1 = c1.newInstance();

Method[] declared\_Methods = c1.getDeclaredMethods();

for (Method method : declared\_Methods) {

String methodName = method.getName();

if (methodName.contains("main"))

continue;

System.out.format("Invoking %s()%n", methodName);

try {

method.setAccessible(true);

Object returnValue = method.invoke(t1, "");

System.out.format("%s() returned: %d%n", methodName, returnValue);

} catch (InvocationTargetException ex) {

System.err.println("An InvocationTargetException was caught!");

Throwable cause = ex.getCause();

System.out.format("Invocation of %s failed because of: %s%n",

methodName, cause.getMessage());

}

}

} catch (ClassNotFoundException | InstantiationException | IllegalAccessException ex) {

System.err.println("The following exception was thrown:");

ex.printStackTrace();

}

}

}

Output:

Invoking testMethod()

An InvocationTargetException was caught!

Invocation of testMethod failed because of: The string must contain at least one character!

Output: This code will generate an InstantiationException.

### **Unchecked exception**

The unchecked exceptions are those exceptions that occur during the execution of the program. Hence they are also referred to as **Runtime exceptions**. These exceptions are generally ignored during the compilation process. They are not checked while compiling the program. For example, programming bugs like logical errors, and using incorrect APIs.

To illustrate the concept of an unchecked exception, let us consider the following code snippet:

import java.util.Scanner;

public class Sample\_RunTimeException {

public static void main(String[] args) {

// Reading user input

Scanner input\_dev = new Scanner(System.in);

System.out.print("Enter your age in Numbers: ");

int age1 = input\_dev.nextInt();

if (age1>20) {

System.out.println("You can view the page");

} else {

System.out.println("You cannot view the page");

}

}

}

Output 1:

Enter your age in Numbers: 21

You can view the page

Output 2:

Enter your age in Numbers: Twelve

Exception in thread “main” java.util.InputMismatchException

at java.util.Scanner.throwFor (Unknown Source)

at java.util.Scanner.next (Unknown Source)

at java.util.Scanner.nextInt (Unknown Source)

at java.util.Scanner.nextInt (Unknown Source)

at exceptiondemo.sample\_runtimedemo.main(Sample\_RunTimeExceptionDemo.java:11)

Now, let us learn about other unchecked exceptions. Some of them are:

#### **NullPointerException**

This type of exception occurs when you try to access an object with the help of a reference variable whose current value is null or empty. For example, consider the following code snippet:

// Program to demonstrate the NullPointerException

class SampleNullPointer {

public static void main(String args[]) {

try {

String a1 = null; // null value

System.out.println(a1.charAt(0));

} catch(NullPointerException e) {

System.out.println("NullPointerException is found in the program.");

}

}

}

Output: NullPointerException is found in the program.

#### **ArrayIndexOutofBound**

This type of exception occurs when you try to access an array with an invalid index value. The value you are providing is either negative or beyond the length of the array.

For example, consider the following code snippet:

// Program to demonstrate the ArrayIndexOutOfBoundException

class sample\_ArrayIndexOutOfBound {

public static void main(String args[]) {

try {

int b[] = new int[6];

b[8] = 2; // we are trying to access 9th element in an array of size 7

} catch(ArrayIndexOutOfBoundsException e) {

System.out.println ("The array index is out of bound");

}

}

}

Output: The array index is out of bound

#### **IllegalArgumentException**

This type of exception occurs whenever an inappropriate or incorrect argument is passed to a method. For example, if a method is defined with non-empty string as parameters. But you are providing null input strings. Then, the [IllegalArgumentException](http://docs.oracle.com/javase/7/docs/api/java/lang/IllegalArgumentException.html) is thrown to indicate the user that you cannot pass a null input string to the method.

Consider the following code snippet to demonstrate this type of exception:

import java.io.File;

public class Sample\_IllegalArgumentException {

public static String createRelativePath(String par, String f\_name) {

if (par == null)

throw new IllegalArgumentException("You cannot provide null parent path!");

if (f\_name == null)

throw new IllegalArgumentException("Please enter the complete filename!");

return par + File.separator + f\_name;

}

public static void main(String[] args) {

// This command will be successfully executed.

system.out.println(IllegalArgumentExceptionExample.createRelativePath("dir1", "file1"));

system.out.println();

// This command will throw an IllegalArgumentException.

System.out.println(IllegalArgumentExceptionExample.createRelativePath(null, "file1"));

}

}

Output: This code will generate an IllegalArgumentException.

#### **IllegalStateException**

This type of exception occurs when the state of the environment does not match the operation being executed. For example, consider the following code snippet, which demonstrates this type of exception:

/\*\*

\* This code will publish the current book.

\* If the book is already published, it will throw an IllegalStateException.

\*\*/

public void pub() throws IllegalStateException {

Date pub\_at = getPub\_at();

if (pub\_at == null) {

setPub\_at(new Date());

Logging.log(String.format("Published '%s' by %s.", getTitle(), getAuthor()));

} else {

throw new IllegalStateException(

String.format("Cannot publish '%s' by %s (already published on %s).",

getTitle(), getAuthor(), pub\_at));

}

}

Output: This code will generate IllegalStateException.

If a publication date already exists in the system, then it will produce an IllegalStateException that indicates that the book cannot be published again.

#### **NumberFormatException**

This type of exception occurs when you pass a string to a method that cannot be converted to a number. For example, consider the following code snippet:

// Program to demonstrate the NumberFormatException

class Sample\_NumberFormat {

public static void main(String args[]) {

try {

// "Test" is not a number

int n = Integer.parseInt ("Test") ;

System.out.println(n);

} catch(**NumberFormatException** e) {

System.out.println("Number format exception");

}

}

}

Output: This code will generate NumberFormatException.

#### **ArithmeticException**

This type of exception occurs when you perform an incorrect arithmetic operation. For example, if you divide any number by zero, it will display such an exception. Let us consider the following code snippet:

// Program to demonstrate the ArithmeticException

class Sample\_ArithmeticException {

public static void main(String args[]) {

try {

int p = 30, q = 0;

int r = p/q;  // It cannot be divided by zero

System.out.println ("Result = " + r);

} catch(ArithmeticException e) {

System.out.println ("Number cannot be divided by 0");

}

}

}

Output: This code will generate an ArithmeticException.

# SERVLET

**Servlet** technology is used to create a web application (resides at server side and generates a dynamic web page).

**Servlet** technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was common as a server-side programming language. However, there were many disadvantages to this technology. We have discussed these disadvantages below.

There are many interfaces and classes in the Servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse, etc.

## What is a Servlet?

Servlet can be described in many ways, depending on the context.

* Servlet is a technology which is used to create a web application.
* Servlet is an API that provides many interfaces and classes including documentation.
* Servlet is an interface that must be implemented for creating any Servlet.
* Servlet is a class that extends the capabilities of the servers and responds to the incoming requests. It can respond to any requests.
* Servlet is a web component that is deployed on the server to create a dynamic web page.

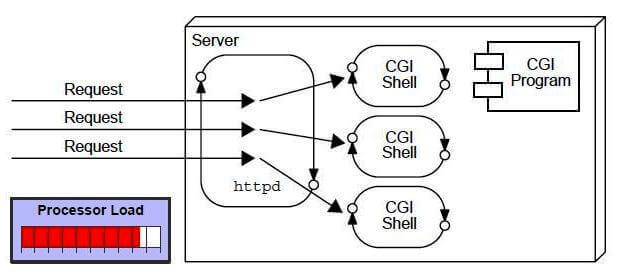
### **What is a web application?**

A web application is an application accessible from the web. A web application is composed of web components like Servlet, JSP, Filter, etc. and other elements such as HTML, CSS, and JavaScript. The web components typically execute in Web Server and respond to the HTTP request.

## Servlet

## CGI (Common Gateway Interface)

CGI technology enables the web server to call an external program and pass HTTP request information to the external program to process the request. For each request, it starts a new process.

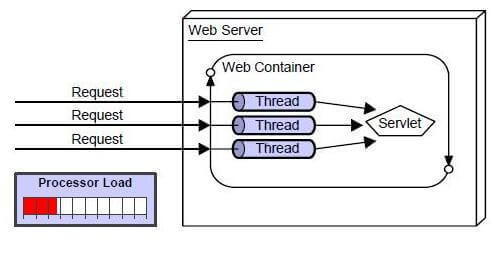


### **Disadvantages of CGI**

There are many problems in CGI technology:

1. If the number of clients increases, it takes more time for sending the response.
2. For each request, it starts a process, and the web server is limited to start processes.
3. It uses platform dependent language e.g. C, C++, perl.

### **Advantages of Servlet**



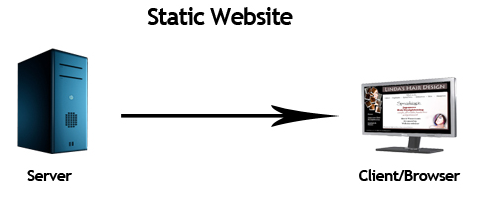
There are many advantages of Servlet over CGI. The web container creates threads for handling the multiple requests to the Servlet. Threads have many benefits over the Processes such as they share a common memory area, lightweight, cost of communication between the threads are low. The advantages of Servlet are as follows:

1. **Better performance:** because it creates a thread for each request, not process.
2. **Portability:** because it uses Java language.
3. **Robust:** JVM manages Servlets, so we don't need to worry about the memory leak, garbage collection, etc.
4. **Secure:** because it uses java language.

## Static website

Static website is the basic type of website that is easy to create. You don't need the knowledge of web programming and database design to create a static website. Its web pages are coded in HTML.

The codes are fixed for each page so the information contained in the page does not change and it looks like a printed page.



## Dynamic website

Dynamic website is a collection of dynamic web pages whose **content changes dynamically**. It **accesses content from a database or Content Management System (CMS).** Therefore, when **you alter or update the content of the database, the content of the website is also altered or updated**.

Dynamic website uses **client-side scripting or server-side scripting, or both to generate dynamic content**.

**Client side scripting generates content at the client computer on the basis of user input**. The web browser downloads the web page from the server and processes the code within the page to render information to the user.

**In server side scripting, the software runs on the server and processing is completed in the server then plain pages are sent to the user.**



## Static vs Dynamic website

|  |  |
| --- | --- |
| **Static Website** | **Dynamic Website** |
| Prebuilt content is same every time the page is loaded. | Content is generated quickly and changes regularly. |
| It uses the **HTML**code for developing a website. | It uses the server side languages such as **PHP,SERVLET, JSP, and ASP.NET**etc. for developing a website. |
| It sends exactly the same response for every request. | It may generate different HTML for each of the request. |
| The content is only changed when someone publishes and updates the file (sends it to the web server). | The page contains "server-side" code which allows the server to generate the unique content when the page is loaded. |
| Flexibility is the main advantage of static website. | Content Management System (CMS) is the main advantage of dynamic website. |

**The Basic Characteristics of HTTP (Hyper Text Transfer Protocol):**

* It is the protocol that allows web servers and browsers to exchange data over the web.
* It is a request response protocol.
* It uses the reliable TCP connections by default on TCP port 80.
* It is stateless means each request is considered as the new request. In other words, server doesn't recognize the user by default.

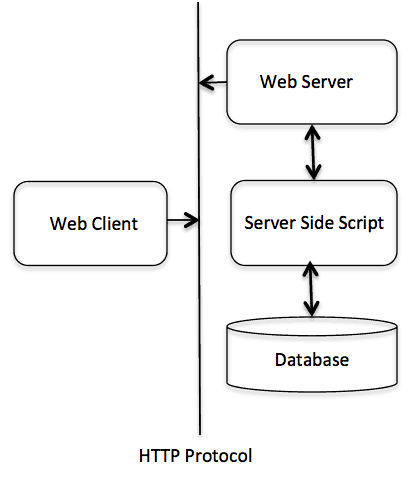
**The Basic Features of HTTP (Hyper Text Transfer Protocol):**

There are three fundamental features that make the HTTP a simple and powerful protocol used for communication:

* **HTTP is media independent:** It specifies that any type of media content can be sent by HTTP as long as both the server and the client can handle the data content.
* **HTTP is connectionless:** It is a connectionless approach in which HTTP client i.e., a browser initiates the HTTP request and after the request is **sent the client disconnects from server and waits for the response.**
* **HTTP is stateless:** The client and server are aware of each other during a current request only. Afterwards**, both of them forget each other**. Due to the stateless nature of protocol, neither the client nor the server can retain the information about different request across the web pages.

**The Basic Architecture of HTTP (Hyper Text Transfer Protocol):**

The below diagram represents the basic architecture of web application and depicts where HTTP stands:



HTTP is request/response protocol which is based on client/server based architecture. In this protocol, web browser, search engines, etc. behave as HTTP clients and the Web server like Servlet behaves as a server

## GET and POST

**Two common methods for the request-response between a server and client are:**

* **GET**- It requests the data from a specified resource
* **POST**- It submits the processed data to a specified resource

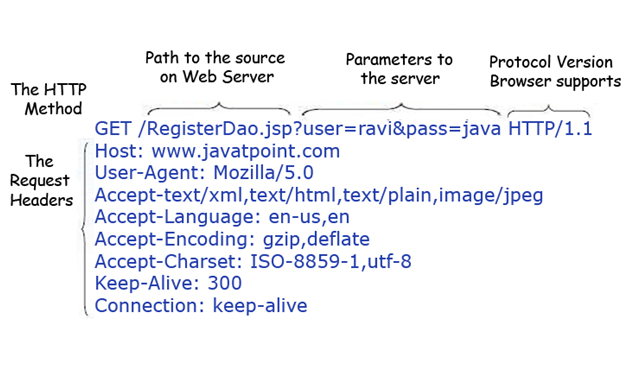
|  |  |
| --- | --- |
| **GET** | **POST** |
| 1) In case of Get request, only **limited amount of data**can be sent because data is sent in header. | In case of post request, **large amount of data**can be sent because data is sent in body. |
| 2) Get request is **not secured**because data is exposed in URL bar. | Post request is **secured**because data is not exposed in URL bar. |
| 3) Get request **can be bookmarked.** | Post request **cannot be bookmarked.** |
| 4) Get request is **idempotent**. It means second request will be ignored until response of first request is delivered | Post request is **non-idempotent.** |
| 5) Get request is **more efficient**and used more than Post. | Post request is **less efficient**and used less than get. |

## Anatomy of Get Request

The query string (name/value pairs) is sent inside the URL of a GET request:

1. GET/RegisterDao.jsp?name1=value1&name2=value2

As we know that data is sent in request header in case of get request. It is the default request type. Let's see what information is sent to the server.

1. 

## Anatomy of Post Request

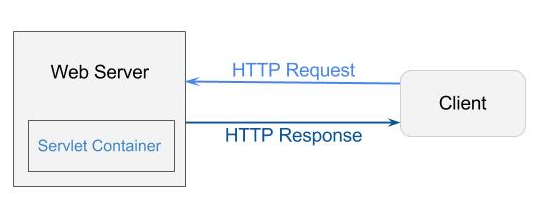
The query string (name/value pairs) is sent in HTTP message body for a POST request:

1. POST/RegisterDao.jsp HTTP/1.1
2. Host: www. javatpoint.com
3. name1=value1&name2=value2
4. As we know, in case of post request original data is sent in message body. Let's see how information is passed to the server in case of post request.
5. 

## Servlet Container

It provides the runtime environment for JavaEE (j2ee) applications. The client/user can request only a static WebPages from the server. If the user wants to read the web pages as per input then the servlet container is used in java.

The servlet container is the part of web server which can be run in a separate process. We can classify the servlet container states in three types:



**Servlet Container States**

The servlet container is the part of web server which can be run in a separate process. We can classify the servlet container states in three types:

* **Standalone:** It is typical Java-based servers in which the servlet container and the web servers are the integral part of a single program. For example:- Tomcat running by itself
* **In-process:** It is separated from the web server, because a different program runs within the address space of the main server as a plug-in. For example:- Tomcat running inside the JBoss.
* **Out-of-process:** The web server and servlet container are different programs which are run in a different process. For performing the communications between them, web server uses the plug-in provided by the servlet container.

**The Servlet Container performs many operations that are given below:**

* Life Cycle Management
* Multithreaded support
* Object Pooling
* Security etc.

[**next →**](https://www.javatpoint.com/content-type)[**← prev**](https://www.javatpoint.com/container)

## Server: Web vs. Application

Server is a device or a computer program that accepts and responds to the request made by other program, known as client. It is used to manage the network resources and for running the program or software that provides services.

There are two types of servers:

1. Web Server
2. Application Server

**Web Server**

Web server contains only web or servlet container. It can be used for servlet, jsp, struts, jsf etc. It can't be used for EJB.

It is a computer where the web content can be stored. In general web server can be used to host the web sites but there also used some other web servers also such as FTP, email, storage, gaming etc.

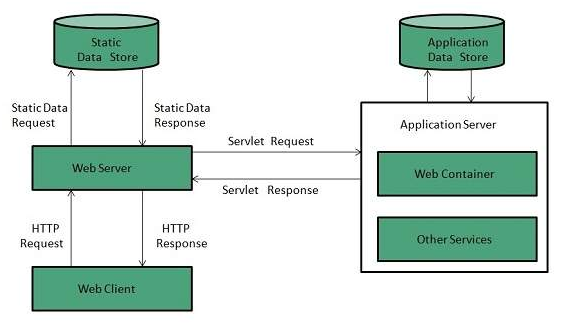
Examples of Web Servers are: **Apache Tomcat**and **Resin**.

## Web Server Working

It can respond to the client request in either of the following two possible ways:

* Generating response by using the script and communicating with database.
* Sending file to the client associated with the requested URL.

The block diagram representation of Web Server is shown below:



**Important points**

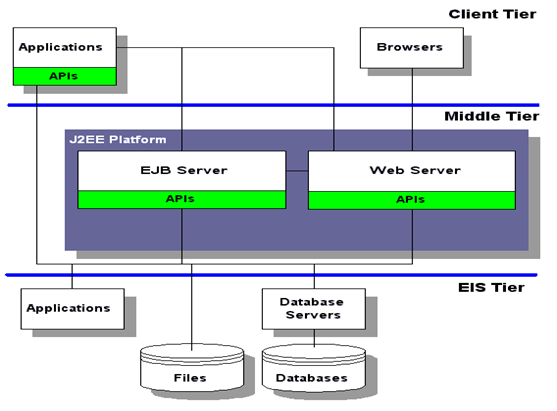
* If the requested web page at the client side is not found, then web server will sends the HTTP response: Error 404 Not found.
* When the web server searching the requested page if requested page is found then it will send to the client with an HTTP response.
* If the client requests some other resources then web server will contact to application server and data is store for constructing the HTTP response.

**Application Server**

Application server contains Web and EJB containers. It can be used for servlet, jsp, struts, jsf, ejb etc. It is a component based product that lies in the middle-tier of a server centric architecture.

It provides the middleware services for state maintenance and security, along with persistence and data access. It is a type of server designed to install, operate and host associated services and applications for the IT services, end users and organizations.

The block diagram representation of Application Server is shown below:



The Example of Application Servers are:

1. **JBoss:** Open-source server from JBoss community.
2. **Glassfish:** Provided by Sun Microsystem. Now acquired by Oracle.
3. **Weblogic:** Provided by Oracle. It more secured.
4. **Websphere:** Provided by IBM

## Servlet API

The javax.servlet and javax.servlet.http packages represent interfaces and classes for servlet api.

The **javax.servlet** package contains many interfaces and classes that are used by the servlet or web container. These are not specific to any protocol.

The **javax.servlet.http** package contains interfaces and classes that are responsible for http requests only.

Let's see what are the interfaces of javax.servlet package.

### **Interfaces in javax.servlet package**

There are many interfaces in javax.servlet package. They are as follows:

1. Servlet
2. ServletRequest
3. ServletResponse
4. RequestDispatcher
5. ServletConfig
6. ServletContext
7. SingleThreadModel
8. Filter
9. FilterConfig
10. FilterChain
11. ServletRequestListener
12. ServletRequestAttributeListener
13. ServletContextListener
14. ServletContextAttributeListener

### **Classes in javax.servlet package**

There are many classes in javax.servlet package. They are as follows:

1. GenericServlet
2. ServletInputStream
3. ServletOutputStream
4. ServletRequestWrapper
5. ServletResponseWrapper
6. ServletRequestEvent
7. ServletContextEvent
8. ServletRequestAttributeEvent
9. ServletContextAttributeEvent
10. ServletException
11. UnavailableException

### **Interfaces in javax.servlet.http package**

There are many interfaces in javax.servlet.http package. They are as follows:

1. HttpServletRequest
2. HttpServletResponse
3. HttpSession
4. HttpSessionListener
5. HttpSessionAttributeListener
6. HttpSessionBindingListener
7. HttpSessionActivationListener
8. HttpSessionContext (deprecated now)

### **Classes in javax.servlet.http package**

There are many classes in javax.servlet.http package. They are as follows:

1. HttpServlet
2. Cookie
3. HttpServletRequestWrapper
4. HttpServletResponseWrapper
5. HttpSessionEvent
6. HttpSessionBindingEvent
7. HttpUtils (deprecated now)

## Servlet Interface

1. [Servlet Interface](https://www.javatpoint.com/Servlet-interface)
2. [Methods of Servlet interface](https://www.javatpoint.com/Servlet-interface#servletmethods)

**Servlet interface provides** commonbehaviorto all the servlets.Servlet interface defines methods that all servlets must implement.

Servlet interface needs to be implemented for creating any servlet (either directly or indirectly). It provides 3 life cycle methods that are used to initialize the servlet, to service the requests, and to destroy the servlet and 2 non-life cycle methods.

### **Methods of Servlet interface**

There are 5 methods in Servlet interface. The init, service and destroy are the life cycle methods of servlet. These are invoked by the web container.

|  |  |
| --- | --- |
| **Method** | **Description** |
| **public void init(ServletConfig config)** | initializes the servlet. It is the life cycle method of servlet and invoked by the web container only once. |
| **public void service(ServletRequest request,ServletResponse response)** | provides response for the incoming request. It is invoked at each request by the web container. |
| **public void destroy()** | is invoked only once and indicates that servlet is being destroyed. |
| **public ServletConfig getServletConfig()** | returns the object of ServletConfig. |
| **public String getServletInfo()** | returns information about servlet such as writer, copyright, version etc. |

### **Servlet Example by implementing Servlet interface**

Let's see the simple example of servlet by implementing the servlet interface.

### It will be better if you learn it after visiting steps to create a servlet.

*File: First.java*

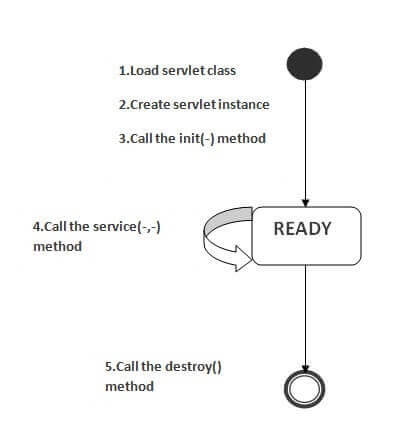
1. **import** java.io.\*;
2. **import** javax.servlet.\*;
4. **public** **class** First **implements** Servlet{
5. ServletConfig config=**null**;
7. **public** **void** init(ServletConfig config){
8. **this**.config=config;
9. System.out.println("servlet is initialized");
10. }
12. **public** **void** service(ServletRequest req,ServletResponse res)
13. **throws** IOException,ServletException{
15. res.setContentType("text/html");
17. PrintWriter out=res.getWriter();
18. out.print("<html><body>");
19. out.print("<b>hello simple servlet</b>");
20. out.print("</body></html>");
21. }
22. **public** **void** destroy(){System.out.println("servlet is destroyed");}
23. **public** ServletConfig getServletConfig(){**return** config;}
24. **public** String getServletInfo(){**return** "copyright 2007-1010";}
26. }

## Life Cycle of a Servlet (Servlet Life Cycle)

1. [Life Cycle of a Servlet](https://www.javatpoint.com/life-cycle-of-a-servlet)
   1. [Servlet class is loaded](https://www.javatpoint.com/life-cycle-of-a-servlet#servletlifecycle1)
   2. [Servlet instance is created](https://www.javatpoint.com/life-cycle-of-a-servlet#servletlifecycle2)
   3. [init method is invoked](https://www.javatpoint.com/life-cycle-of-a-servlet#servletlifecycle3)
   4. [service method is invoked](https://www.javatpoint.com/life-cycle-of-a-servlet#servletlifecycle4)
   5. [destroy method is invoked](https://www.javatpoint.com/life-cycle-of-a-servlet#servletlifecycle5)

The web container maintains the life cycle of a servlet instance. Let's see the life cycle of the servlet:

1. Servlet class is loaded.
2. Servlet instance is created.
3. init method is invoked.
4. service method is invoked.
5. destroy method is invoked.



As displayed in the above diagram, there are three states of a servlet: new, ready and end. The servlet is in new state if servlet instance is created. After invoking the init() method, Servlet comes in the ready state. In the ready state, servlet performs all the tasks. When the web container invokes the destroy() method, it shifts to the end state.

### **1) Servlet class is loaded**

The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

### **2) Servlet instance is created**

The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

### **3) init method is invoked**

|  |
| --- |
| The web container calls the init method only once after creating the servlet instance. The init method is used to initialize the servlet. It is the life cycle method of the javax.servlet.Servlet interface. Syntax of the init method is given below: |

1. **public** **void** init(ServletConfig config) **throws** ServletException

### **4) service method is invoked**

The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method. If servlet is initialized, it calls the service method. Notice that servlet is initialized only once. The syntax of the service method of the Servlet interface is given below:

1. **public** **void** service(ServletRequest request, ServletResponse response)
2. **throws** ServletException, IOException

### **5) destroy method is invoked**

The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc. The syntax of the destroy method of the Servlet interface is given below:

1. **public** **void** destroy()

## load on startup in web.xml

The **load-on-startup** element of **web-app** loads the servlet at the time of deployment or server start if value is positive. It is also known as **pre initialization of servlet**.

You can pass positive and negative value for the servlet.

#### **Advantage of load-on-startup element**

As you know well, servlet is loaded at first request. That means it consumes more time at first request. If you specify the load-on-startup in web.xml, servlet will be loaded at project deployment time or server start. So, it will take **less time** for responding to first request.

#### **Passing positive value**

If you pass the positive value, the lower integer value servlet will be loaded before the higher integer value servlet. In other words, container loads the servlets in ascending integer value. The 0 value will be loaded first then 1, 2, 3 and so on.

Let's try to understand it by the example given below:

*web.xml*

1. **<web-app>**
2. ....
4. **<servlet>**
5. **<servlet-name>**servlet1**</servlet-name>**
6. **<servlet-class>**com.javatpoint.FirstServlet**</servlet-class>**
7. **<load-on-startup>**0**</load-on-startup>**
8. **</servlet>**
10. **<servlet>**
11. **<servlet-name>**servlet2**</servlet-name>**
12. **<servlet-class>**com.javatpoint.SecondServlet**</servlet-class>**
13. **<load-on-startup>**1**</load-on-startup>**
14. **</servlet>**
16. ...
17. **</web-app>**

There are defined 2 servlets, both servlets will be loaded at the time of project deployment or server start. But, servlet1 will be loaded first then servlet2.

#### **Passing negative value**

If you pass the negative value, servlet will be loaded at request time, at first request.

# A Servlet tutorial from <https://www.java4s.com/java-servlet-tutorials/what-is-servlet-an-introduction-to-java-servlets/>

## What is servlet? An Introduction to Java Servlets

[Servlets](https://www.java4s.com/java-servlet-tutorials/) » on Jan 4, 2013 [**{ 11 Comments }**](https://www.java4s.com/java-servlet-tutorials/what-is-servlet-an-introduction-to-java-servlets/#comments) By Sivateja

The basic aim of servlet is to develop web applications.  Before servlets came into picture there was a specification called CGI.  Servlets specification developed by SUN and released to the industry.   Lot of server vendors came forward for implementing these specifications of servlets.  Servlets are nothing but set of rules given by SUN in the form of Interfaces.  The server vendors have developed their own classes by implementing the interfaces developed by SUN.

The collection of classes developed by server vendors and Interfaces developed by SUN are known as Servlet-API.

## What is servlet

A servlet is a server side platform independent, dynamic and multithread java program, which runs in the context of server for extending the functionality of server.  When multiple users make a request to the same servlet then all requests will be processed by the container by creating multiple threads for the same servlet. In order to deal with these servlet programming, we must import

12javax.servlet.\*;

javax.servlet.http.\*;

into our java program.

|  |  |
| --- | --- |
|  |  |
|  |  |

* javax.servlet.Servlet is one of pre-defined top most interface, which is containing life cycle methods[init(), service(), destroy() i will describe about these methods later] for servlet.
* javax.servlet.GenericServlet is one of the predefined abstract class which is implementing javax.servlet.Servlet interface,  this GenericServlet class **is used for developing protocol independent applications** [ **But in real time we will use protocol dependent applications only, so GenericServlet won’t be appeared in real time** ]
* Javax.servlet.http.HttpServlet is sub class of GenericServlet used for **developing protocol dependent application**  [**HTTP** protocol ]

Consider Myserlet.java is our own class always recommended to extends HttpServlet servlet, as our current internet world is supporting HTTP protocol right 🙂  In real time, we will use HttpServlet only not GenericServlet remember.

## 

## Steps to Write Java Servlet Program

[Servlets](https://www.java4s.com/java-servlet-tutorials/) » on Jan 5, 2013 [**{ 6 Comments }**](https://www.java4s.com/java-servlet-tutorials/steps-to-write-java-servlet-program/#comments) By Sivateja

Let us see the basic steps to develop java servlet application…

Servlet program is not like, writing java code and execute through command prompt.  We need to follow the following steps in order to develop any servlets program.  Even for a simple HelloWorld program also one must follow this standard directory structure which is prescribed by sun.

* Create one root directory with your own name, and create another directory within that directory with name ‘src’ and now write one servlet program and copy into that ‘src’ folder
* Now create another folder ‘web-inf’ in the root directory, this web-inf folder contains web.xml file will see later
* Create ‘**classes’ folder with in web-inf** folder
* Compile our servlet program which is in src folder, so we will get one .class file right, just copy this .class file into classes folder in web-inf folder
* By the way, we are going to run our servlet by using Tomcat server.  So we must set the class path for Tomcat related jar file, servlet-api.jar
* You can find this jar file in your Tomcat directory\common\lib\,
* Similarly if we are using Weblogic server, we need to set the class path for weblogic related jar files, i.e weblogic.jar which is in c [your drive]:\bea\weblogic\server\lib
* If our servlet is dealing with any database like MySql or Oracle, it is mandatory for us to create a folder  ‘lib’ with in web-inf folder
* Once everything is done, just copy our root director into Tomcat/webapps folder, and start the server and check the output, i will show you how in the first application

## Steps To Develop Servlet Program

* Set servlet-api.jar ( Will be available in your tomcat/lib folder ) in your class path
* Open notepad or any text editor EditPlus or some thing
* Import javax.servlet.\*;
* Import javax.servlet.http.\*;
* Create one user defined class whose modifier must be public, which must extends javax.servlet.GenericServlet[Not recommended] or  javax.servlet.http.HttpServlet[**recommended**]
* We know GenericServlet implementing Servlet interface,  and HttpServlet is the sub class of GenericServlet.  So by default all life cycle methods will be overridden in GenericServlet class, so just override the required [No need all] life cycle methods of servlet in HttpServlet.
* Save your servlet program by giving an extension .java
* You must place your .java files in root folder or in src folder, and .class files in classes files
* Compile and check the result

Yeah these are the basic and default steps for writing any servlet application, i will tell you the more(web.xml, deployments and all…) while we are working with the examples 😉

Once everything is done, just copy our root director into Tomcat/webapps folder, and start the server and check the output, i will show you in the first example.

## How flow goes url to web xml to Write Deployment Descriptor,web.xml In Servlet

[Servlets](https://www.java4s.com/java-servlet-tutorials/) » on Jan 5, 2013 [**{ 22 Comments }**](https://www.java4s.com/java-servlet-tutorials/how-to-write-deployment-descriptorweb-xml-in-servlet/#comments) By Sivateja

This web.xml is like index of book, web.xml is containing details of static web resource programs and dynamic web resource programs.  The purpose of web.xml is to hide [or] to achieve the security for the web application by not showing [ in the URL ] the type of technology used for development of web application.

## Deployment Descriptor [ web.xml ] in Servlets

## Syntax

12345678910<web-app>

<servlet>

<servlet-name> Give Some Dummy Name </servlet-name>

<servlet-class>/Flly qualified name of our servlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name> Give Some Dummy Name </servlet-name>

<url-pattern>/UserFriendlyName</url-pattern>

</servlet-mapping>

</web-app>

* We must call our servlet by typing its <url-pattern> only
* For example http://localhost:8080/OurServletApplicationFolderName/UserFriendlyName
* Once we call our servlet just like above, then the server will first loads web.xml and verifies whether the url pattern we are calling is same as what in <url-pattern> tag in web.xml [Line number 8] or not
* **If matched**, it will checks the <servlet-name> [Line number 7] and will jump to <servlet-name> [Line number 3] in <servlet> tag and, will check whether the value of <servlet-name> in line number 3 & line number 7 are same or not **if same it will load our class in <servlet-class>** [line number 4]
* Finally our .class file will be executed, so we can check the output

## Servlet Hello World Example in Eclipse IDE with Tomcat Server

[Servlets](https://www.java4s.com/java-servlet-tutorials/) » on Jan 5, 2013 [**{ 5 Comments }**](https://www.java4s.com/java-servlet-tutorials/servlet-hello-world-example-in-eclipse-ide-with-tomcat-server/#comments) By Sivateja

We will see the first servlet application directly in Eclipse IDE and i am using Tomcat 7, friends doing java servlet with out eclipse is really tedious 😉 we cannot do every thing individually and copying related files into the folders and finally copying the application into tomcat webapps directory bla bla.., So we can do it very easily in Eclipse IDE will see how.

Open Eclipse IDE [ Download any version from Google ], Goto File > New > Dynamic web Project

We should write .java files in src folder and .class files will be automatically copied into build folder including the package, so i am going to create .java file OnGeneric.java [ First application i am going to tell you using GenericServlet, from the next application i will take HttpServlet only ]

## OnGeneric.java Before setting servlet-api.jar

So we need to create the classpath for servlet-api.jar > right click on ROOT folder [ Servlet-Hello-World-Java4s ] > Properties > now a window will open in front of you > left side click on Java Build Path > and click on Add External Jars and add the servlet-api.jar from the tomcat folder > and click on ok

|  |  |
| --- | --- |
| https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif | https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif |

That’s it,  check now all the errors will be gone 🙂

|  |  |
| --- | --- |
| https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif | https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif |

Now its the time to run our program, for that we should add the server [i will tell you about web.xml later], So in the server window right click and New > Server

|  |  |
| --- | --- |
| https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif | https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif |

Choose tomcat 7 > Next >

|  |  |
| --- | --- |
| https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif | https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif |

Click on Browse > choose Tomcat installation directory > Next

|  |  |
| --- | --- |
| https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif | https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif |

Choose our existing project at left side and click on Add and Finish

|  |  |
| --- | --- |
| https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif | https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif |

Now right click on project [Root] > Run As > Run on Server > Output

|  |  |
| --- | --- |
| https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif | https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif |

See in the URL ‘Servlet-Hello-World-Java4s’ is our servlet application directory, now come to web.xml

|  |  |
| --- | --- |
| https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif | https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif |

We should give the url pattern in the url to get the output…

|  |  |
| --- | --- |
| https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif | https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif |

## Servlet Hello World Application

Files required….

* OnGeneric.java
* web.xml

## Directory Structure

|  |  |
| --- | --- |
| https://www.java4s.com/wp-content/themes/strPro4Tut/java4s/bg00000.gif |  |

## OnGeneric.java

123456789101112131415161718192021package java4s;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.GenericServlet;

import javax.servlet.ServletException;

import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

public class OnGeneric extends GenericServlet

{

public void service(ServletRequest req,ServletResponse res)throws ServletException,IOException

{

res.setContentType("text/html");

PrintWriter pw=res.getWriter();

pw.println("This is first servlet program on GenericServlet\_\_\_Java4s.com");

pw.close();

}

}

## Explanation

* We must import required packages javax.servlet.\*; and java.io.\*
* And our class should extend GenericServlet or HttpServlet, i am using GenericServlet in this application
* We **must over ride service(-,-)** method in our class having request and response object
* **res.setContentType(); means we are setting what type of response we are sending back**
* **PrintWriter will be used to write the response, we can get the PrintWriter object from the response object [ res.getWriter() ]**
* **That’s it, once we call close() method of PrintWriter, then response will be sent back**

## Web.xml

<web-app>

<servlet>

<servlet-name>first</servlet-name>

<servlet-class>java4s.OnGeneric</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>first</servlet-name>

<url-pattern>/firstHelloWorld</url-pattern>

</servlet-mapping>

</web-app>

Output i shown you already 🙂

## Methods in javax.servlet.http.HttpServlet

[Servlets](https://www.java4s.com/java-servlet-tutorials/) » on Jan 6, 2013 [{ 1 Comment }](https://www.java4s.com/java-servlet-tutorials/methods-in-javax-servlet-http-httpservlet/#comments) By Sivateja

As we discussed in the earlier articles, we should **extend our servlet class with HttpServlet abstract class in order to get protocol dependent [http] services**. When ever we make request to the servlet the following methods will be called automatically by the container.

* Container first calls public **Service()** method
* Public Service() method internally calls protected service method
* Protected Service() method will internally calling doGet() or doPost() [ i mean doXxx() methods, its depends on the type of http method used by the client ]
* If the client is not specifying the type of Http method then **Http protocol by default consider GET method**, so finally the client request is processed at doGet() method
* In general when we are developing any servlets we must take a user defined class are it must extends HttpServlet and override either doGet() or doPost() or Both.

## Servlet Life Cycle in Java, Explanation of Servlet Life Cycle Methods

[Servlets](https://www.java4s.com/java-servlet-tutorials/) » on Jan 6, 2013 [**{ 18 Comments }**](https://www.java4s.com/java-servlet-tutorials/servlet-life-cycle-in-java-explanation-of-servlet-life-cycle-methods/#comments) By Sivateja

Aware of servlet life cycle is very important, before you going to execute first application. So let us see flow of life cycle methods…

Java servlet will be executed in different stages, of which

* Loading the application and its Instantiation, Initialization
* Executing the business logic
* Finally destroying

|  |  |
| --- | --- |
|  |  |
|  |  |

Let us see whats going in these stages 🙂

We know that, **javax.servlet.Servlet is the super interface**, having the methods…

* init(ServletConfig config)
* **service**(ServletRequest req,ServletResponse res)throws ServletException,IOException
* destroy()

## Loading the application and its Instantiation,Initialization

In the first request, the container will creates the object of the **servelt and executes the init() method first, remember this init() method will be executed only once…**

* At the time of giving the first request
* If we **use ‘load-on-startup‘ tag in the web.xml** then, our servelt will be loaded and init() method will be executed at the time of server starts, will see about this ‘load-on-startup‘ in the upcoming articles
* In general**, database connection opening or any one time object creation tasks will be written in this method, as init() will be executed only once**, hope you got one idea 😉

## Executing the business logic

So business logic, which is in service(-,-) method, when ever we send the request to the servlet, **every time this service() method will be executed.**  If you send 100 requests to the servlet, 100 times service(-,-) method will be executed but init() method ? only one time :-).  This service() method internally calls doGet(-,-) or doPost(-,-) methods in HttpServlet, i will explain about this in the examples later, just remember as of now.

## Finally destroying

This destroy() method will be called before servlet is removed from the container, and finally it will be garbage collected as usual.

Friends, just remember all this concept, i will explain line by line execution flow in the first servlet program, there i will touch every thing for you again 🙂

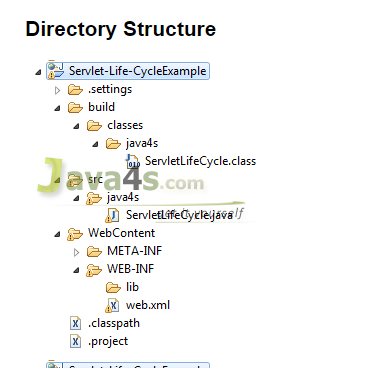
## Example on Servlet Life Cycle in Java

[Servlets](https://www.java4s.com/java-servlet-tutorials/) » on Jan 10, 2013 [**{ 16 Comments }**](https://www.java4s.com/java-servlet-tutorials/example-on-servlet-life-cycle-in-java/#comments) By Sivateja

Let us see one example on servlet life cycle. Do you have any doubts related to the concept ? if so please check this [Servlet Life Cycle theory](https://www.java4s.com/java-servlet-tutorials/servlet-life-cycle-in-java-explanation-of-servlet-life-cycle-methods/) article before you execute this example.

### Files required

* ServletLifeCycle.java
* web.xml



### ServletLifeCycle.java

package java4s;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletConfig;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class ServletLifeCycle extends HttpServlet

{

public ServletLifeCycle()

{

System.out.println("Am from default constructor");

}

public void init(ServletConfig config)

{

System.out.println("Am from Init method...!");

}

public void doGet(HttpServletRequest req,HttpServletResponse res)throws ServletException,IOException

{

res.setContentType("text/html");

PrintWriter pw = res.getWriter();

pw.println("I am from doGet method");

pw.close();

}

public void destroy()

{

System.out.println("Am from Destroy methods");

}

}

### web.xml

123456789101112<web-app>

<servlet>

<servlet-name>second</servlet-name>

<servlet-class>java4s.ServletLifeCycle</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>second</servlet-name>

<url-pattern>/lifecycle1</url-pattern>

</servlet-mapping>

</web-app>

### Execution Flow

* Right click on your project ‘root(project folder)’ > Run as > click on ‘Run on Server‘
* Then eclipse will open an internal web browser with URL  —  http://localhost:2013/Servlet-Life-CycleExample/
* Now just append our project folder name to that URL, so the final URL will be like http://localhost:2013/Servlet-Life-CycleExample/Servlet-Life-CycleExample
* Press enter, now you will be able to see the output

### Flow

* When ever you hit the final URL (http://localhost:2013/Servlet-Life-CycleExample/Servlet-Life-CycleExample), servlet container will first loads web.xml and check the value in <url-pattern> tag [ line number 10, we have given the pattern name as ***/lifecycle1*** ]
* Immediately it will check the value in <servlet-name> tag [line number 9, we have given ***second***]
* Now it will search for the same value ***second***in <servlet-name> under <servlet> tag [that’s what we have in line number 3 ]
* So value of <servlet-name> under <servlet-mapping> & <servlet> are same [in this case @ line numbers 9,3 ]
* If so container will load java class given in <servlet-class> tag [line number 4, our class name is java4s.ServletLifeCycle ]
* Finally servlet container came to ‘ServletLifeCycle‘ class, it will executes init() method first (only once) > then service() or doGet() or doPost() method [ in our case doGet() ]
* Finally destroy() method will be executed before our object get garbage collected.

Hope i explained properly, you understand right :-), i will explain <load-on-startup> later just forget as of now.

### Output



## How to Retrieve Client Input Data in Servlet

[Servlets](https://www.java4s.com/java-servlet-tutorials/) » on Jan 14, 2013 [**{ 5 Comments }**](https://www.java4s.com/java-servlet-tutorials/how-to-retrieve-client-input-data-in-servlet/#comments) By Sivateja

When client send some input data to the servlet, that data will be available in the form of request object.  We can retrieve that **input data through HttpServletRequest or ServletRequest interfaces**.  There are 4 approaches to retrieve the client data which is present in request object.

* public String getParameter(“input variable”);
* public Enumeration getParameterNames();
* public Map getParameterMap();
* public String[] getParamterValues(“input variable contains muliple values”);

All these methods are present in javax.servlet.ServletRequest interface, and HttpServletRequest is the sub interface of ServletRequest, ok let us see how to use these methods with an example 🙂

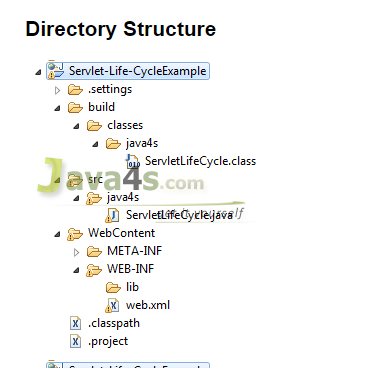
## Example of request.getparameter(), Retrieve Parameters from HTML Form

[Servlets](https://www.java4s.com/java-servlet-tutorials/) » on Jan 14, 2013 [**{ 13 Comments }**](https://www.java4s.com/java-servlet-tutorials/example-of-request-getparameter-retrieve-parameters-from-html-form/#comments) By Sivateja

Let us see how to use request.getParameter() method in the servlet class, to retrieve the input values from HTML page.  Friends it is base concept on retrieving the input data, so observe very carefully, also this is the first example we are seeing on retrieving the values form the input pages.

Hmmm let us calculate sum of two numbers [ by taking these numbers from the input HTML page ]

### Directory Structure



|  |  |
| --- | --- |
|  |  |

### Files Required

* index.html
* OngetParameter.java
* web.xml

### index.html

1234567<font face="verdana" size="2px">

<form action="sum" method="post">

Number1:<input type="text" name="n1"><br>

Number2:<input type="text" name="n2"><br>

<input type="submit" value="Calculate Sum">

</form>

</font>

### OngetParameter.java

package java4s;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class OngetParameter extends HttpServlet

{

protected void doPost(HttpServletRequest req,HttpServletResponse res)throws ServletException,IOException

{

PrintWriter pw=res.getWriter();

res.setContentType("text/html");

String n1=req.getParameter("n1");

String n2=req.getParameter("n2");

int result=Integer.parseInt(n1)+Integer.parseInt(n2);

pw.println("Sum of two numbers " +result);

pw.close();

}

}

### web.xml

<web-app>

<servlet>

<servlet-name>sumOfTwoNumbers</servlet-name>

<servlet-class>java4s.OngetParameter</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>sumOfTwoNumbers</servlet-name>

<url-pattern>/sum</url-pattern>

</servlet-mapping>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

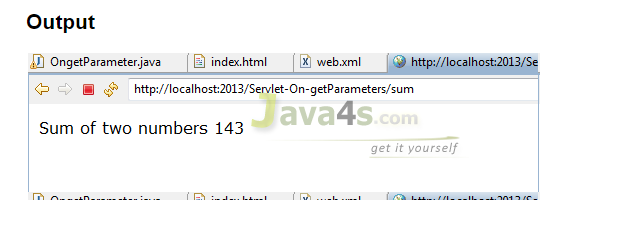
</welcome-file-list>

</web-app>

### Explanation:

* Once we run the application, browser will display index.html as front page.  Because we have given index.html in welcome file list [ line number 14 in web.xml ]
* Enter input values and press ‘Calculate Sum‘ button
* Now come to OngetParameter.java > just retrieve the input values like req.getParameter(“n1”) & req.getParameter(“n2”)
* getParameter() is the method in request object, which returns String value always
* So convert that string output to Integer [ line number 21]
* Integer.parseInt(-String-) gives integer value
* Output

### Output



## Java Servlet login Example In Eclipse

[Servlets](https://www.java4s.com/java-servlet-tutorials/) » on Jan 19, 2013 [**{ 17 Comments }**](https://www.java4s.com/java-servlet-tutorials/java-servlet-login-example-in-eclipse/#comments) By Sivateja

Let us discuss one simple login application using servlet and jsp, friends please refer previous articles if you still have any doubts regarding strvlets flow 🙂

### Files Required

* OnServletLogin.java
* index.html
* web.xml

### index.html

12345678910111213<form action="login" method="post">

<table>

<tr>

<td><font face="verdana" size="2px">Name:</font></td>

<td><input type="text" name="userName"></td>

</tr>

<tr>

<td><font face="verdana" size="2px">Password:</font></td>

<td><input type="password" name="userPassword"></td>

</tr>

</table>

<input type="submit" value="Login">

</form>

### web.xml

123456789101112131415161718<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:web="http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">

<servlet>

<servlet-name>loginServlet</servlet-name>

<servlet-class>java4s.OnServletLogin</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>loginServlet</servlet-name>

<url-pattern>/login</url-pattern>

</servlet-mapping>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

</welcome-file-list>

</web-app>

### OnServletLogin.java

1234567891011121314151617181920212223242526272829package java4s;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class OnServletLogin extends HttpServlet

{

protected void doPost(HttpServletRequest req,HttpServletResponse res)throws ServletException,IOException

{

PrintWriter pw=res.getWriter();

res.setContentType("text/html");

String user=req.getParameter("userName");

String pass=req.getParameter("userPassword");

if(user.equals("java4s")&&pass.equals("java4s"))

pw.println("Login Success...!");

else

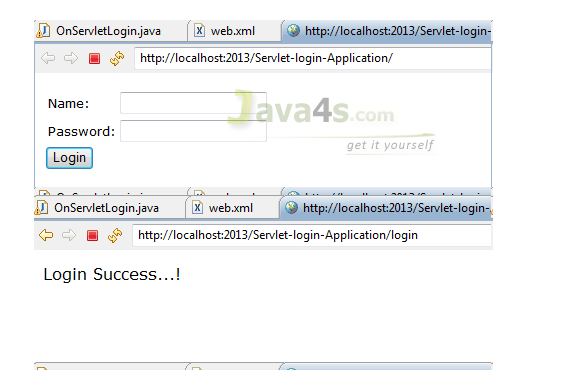
pw.println("Login Failed...!");

pw.close();

}

}

### Output



### Example on getParameterNames() method of Servlet Request Object

[Servlets](https://www.java4s.com/java-servlet-tutorials/) » on Jan 19, 2013 [**{ 4 Comments }**](https://www.java4s.com/java-servlet-tutorials/example-on-getparameternames-method-of-servlet-request-object/#comments) By Sivateja

So far we have worked on [***getParameter()***](https://www.java4s.com/java-servlet-tutorials/example-of-request-getparameter-retrieve-parameters-from-html-form/), let us see how to use getParameterNames() method. With getParameter() we are able to find parameter values by passing parameter name, just like..

In html:  
<input type=”text” name=”n1″>

In Java:  
String val=req.getParameter(“n1”);

Parameter name – n1  
Parameter value – val

If a client send the data to the servlet, that data will be available in the object of HttpServletRequest interface. In case of getParameter() method we have to pass input parameter name and it will give the value. If you are not aware of input parameter name ? or if you have 50+ input values its really tedious to use getParameter() method.

That’s why this getParameterNames() came into picture 🙂

### Files Required

* index.html
* OngetParameterNames.java
* web.xml

### index.html

1234567<font face="verdana" size="2px">

<form action="onPM" method="post">

Name:<input type="text" name="name">

Country:<input type="text" name="country">

<input type="submit" value="Submit">

</form>

</font>

### OngetParameterNames.java

123456789101112131415161718192021222324252627282930package java4s;

import java.io.IOException;

import java.io.PrintWriter;

import java.util.Enumeration;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class OngetParameterNames extends HttpServlet

{

protected void doPost(HttpServletRequest req,HttpServletResponse res)throws ServletException,IOException

{

PrintWriter pw=res.getWriter();

res.setContentType("text/html");

Enumeration en=req.getParameterNames();

while(en.hasMoreElements())

{

Object objOri=en.nextElement();

String param=(String)objOri;

String value=req.getParameter(param);

pw.println("Parameter Name is '"+param+"' and Parameter Value is '"+value+"'");

}

pw.close();

}

}

### web.xml

<web-app>

<servlet>

<servlet-name>onParameterNames</servlet-name>

<servlet-class>java4s.OngetParameterNames</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>onParameterNames</servlet-name>

<url-pattern>/onPM</url-pattern>

</servlet-mapping>

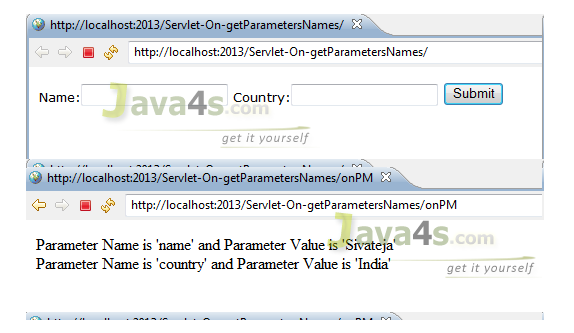
<welcome-file-list>

<welcome-file>index.html</welcome-file>

</welcome-file-list>

</web-app>

### Output



## Example on getParameterMap() method of Servlet Request Object

[Servlets](https://www.java4s.com/java-servlet-tutorials/) » on Jan 27, 2013 [**{ 2 Comments }**](https://www.java4s.com/java-servlet-tutorials/example-on-getparametermap-method-of-servlet-request-object/#comments) By Sivateja

Let us see about getParameterMap() method of servlet request object.  This method is little more useful compared to previous methods [ getParameter(), getParameterNames() ].

### Syntax

Map m = request.getParameterMap()

* getParameterMap() method always returns Map object
* But how we will get input parameter names and their values ? in the form of key and value pairs
* Previous methods [ getParameter(), getParameterNames() ] will be used if there is a chance of getting single value for particular parameter name, something for example check the output screen, there we can give only one value (Sivateja) in Name field right ?
* So what if there is a need of selecting more than one value, for example if you would like your customers to select their habits, here they can have more than one habit right ? so we have to provide check boxes [Means multiple values]

That’s why this getParameterMap() came into picture 🙂

### Files Required

* index.html
* OngetParameterMap.java
* web.xml

### index.html

1234567891011<font face="verdana" size="2px">

<form action="onGPM" method="post">

Name:<input type="text" name="name">

Country:<input type="text" name="country"><br>

Habits : <br>

<input type="checkbox" name="habits" value="Reading">Reading<br>

<input type="checkbox" name="habits" value="Movies">Movies<br>

<input type="checkbox" name="habits" value="Writing">Writing<br>

<input type="submit" value="Submit">

</form>

</font>

### OngetParameterNames.java

12345678910111213141516171819202122232425262728293031323334353637383940414243444546package java4s;

import java.io.IOException;

import java.io.PrintWriter;

import java.util.Iterator;

import java.util.Map;

import java.util.Set;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class OngetParameterMap extends HttpServlet

{

    protected void doPost(HttpServletRequest req,HttpServletResponse res)throws ServletException,IOException

    {

        PrintWriter pw=res.getWriter();

        res.setContentType("text/html");

        Map m=req.getParameterMap();

        Set s = m.entrySet();

        Iterator it = s.iterator();

            while(it.hasNext()){

                Map.Entry<String,String[]> entry = (Map.Entry<String,String[]>)it.next();

                String key             = entry.getKey();

                String[] value     = entry.getValue();

                pw.println("Key is "+key+"<br>");

                    if(value.length>1){

                        for (int i = 0; i < value.length; i++) {

                            pw.println("<li>" + value[i].toString() + "</li><br>");

                        }

                    }else

                            pw.println("Value is "+value[0].toString()+"<br>");

                    pw.println("-------------------<br>");

            }

        pw.close();

    }

}

### Explanation

* Line number 21,  we are getting input parameter values by using getParameterMap() method, which always returns Map object
* So we have all input parameter names and its values in the form of key–value pairs in the Map object, then how to iterate them ?
* We don’t have iterator() method in the Map interface so how ? In the Map interface we have entrySet() method which always returns Set object, so first get that Set object [ that’s what we did at line number 22 ]
* Now iterate that Set object [ line number 23 ]
* line number 27, we are converting the Set generated value into Map.Entry<String,String[]>, means assume one row(Entry) contains key = value ( little imagination will be needed here 😉 )
* Finally i am printing the values, hope you can understand that logic (check the output screen too)

### web.xml

1234567891011121314151617<web-app>

    <servlet>

        <servlet-name>ongetParameterMap</servlet-name>

        <servlet-class>java4s.OngetParameterMap</servlet-class>

    </servlet>

    <servlet-mapping>

            <servlet-name>ongetParameterMap</servlet-name>

            <url-pattern>/onGPM</url-pattern>

    </servlet-mapping>

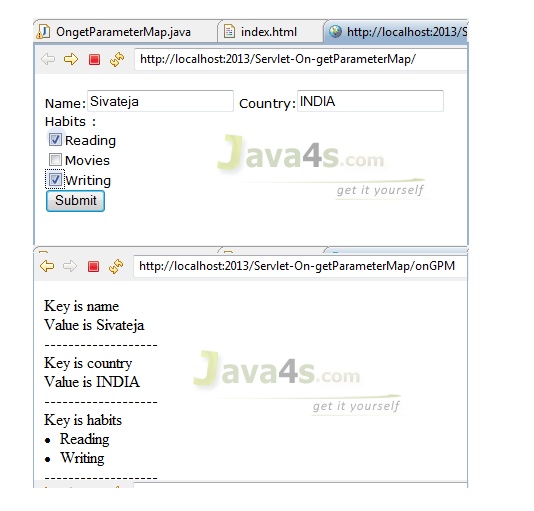
   <welcome-file-list>

        <welcome-file>index.html</welcome-file>

   </welcome-file-list>

</web-app>

### Output



## Example on getParameterValues() method of Servlet Request

[Servlets](https://www.java4s.com/java-servlet-tutorials/) » on Jan 28, 2013 [**{ 9 Comments }**](https://www.java4s.com/java-servlet-tutorials/example-on-getparametervalues-method-of-servlet-request/#comments) By Sivateja

The method getParameterValues() will generally came into picture if there is a chance of getting multiple values for any input parameter, this method will retrieve all of it values and store as string array.

### Syntax

String[] values = getParameterValues(“Input Parameter”);

### Files Required

* index.html
* OngetParameterValues.java
* web.xml

### index.html

12345678910<font face="verdana" size="2px">

<form action="onGPV" method="post">

Habits :

<input type="checkbox" name="habits" value="Reading">Reading

<input type="checkbox" name="habits" value="Movies">Movies

<input type="checkbox" name="habits" value="Writing">Writing

<input type="checkbox" name="habits" value="Singing">Singing

<input type="submit" value="Submit">

</form>

</font>

### OngetParameterNames.java

1234567891011121314151617181920212223242526272829package java4s;

import java.io.IOException;

import java.io.PrintWriter;

import java.util.Iterator;

import java.util.Map;

import java.util.Set;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class OngetParameterValues extends HttpServlet

{

    protected void doPost(HttpServletRequest req,HttpServletResponse res)throws ServletException,IOException

    {

        PrintWriter pw=res.getWriter();

        res.setContentType("text/html");

        String[] values=req.getParameterValues("habits");

        pw.println("Selected Values...");

        for(int i=0;i<values.length;i++)

       {

           pw.println("<li>"+values[i]+"</li>");

       }

       pw.close();

    }

}

### web.xml

1234567891011121314151617<web-app>

    <servlet>

        <servlet-name>ongetParameterValues</servlet-name>

        <servlet-class>java4s.OngetParameterValues</servlet-class>

    </servlet>

    <servlet-mapping>

            <servlet-name>ongetParameterValues</servlet-name>

            <url-pattern>/onGPV</url-pattern>

    </servlet-mapping>

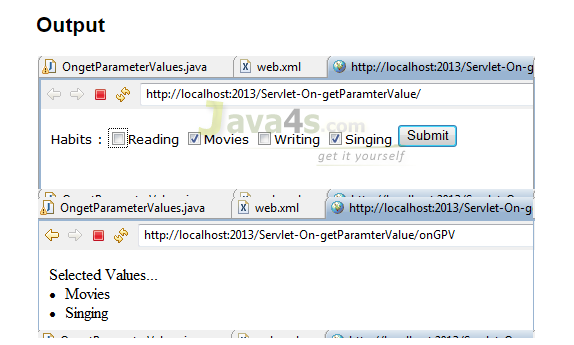
   <welcome-file-list>

        <welcome-file>index.html</welcome-file>

   </welcome-file-list>

</web-app>

### Output



## Understanding ServletConfig and ServletContext

[Servlets](https://www.java4s.com/java-servlet-tutorials/) » on Jan 28, 2013 [**{ 2 Comments }**](https://www.java4s.com/java-servlet-tutorials/understanding-servletconfig-and-servletcontext/#comments) By Sivateja

In our servlet application development, some of the programmers will write fixed coding [ standard lines ], with in the servlet, which is not recommended.  For example, if we write database related code in our servlet, in future if we would like to change any DB credentials in our servlet, we have to open > update > save > compile > re-deploy.  By any chance, what if we need to modify 50+ servlet classes ? its really tedious.

In order to gain the performance of the web application by eliminating the duplicate source code, it is highly recommended to use the fixed technical database details in the web.xml file, web.xml provides external fixed technical details to the servlet.

So in order to develop flexible servlets we make use of web.xml and ServletConfig and ServletContext, let us see one by one in detail.

## Example of ServletConfig in Java Servlet

[Servlets](https://www.java4s.com/java-servlet-tutorials/) » on Jan 28, 2013 [**{ 15 Comments }**](https://www.java4s.com/java-servlet-tutorials/example-of-servletconfig-in-java-servlet/#comments) By Sivateja

Let us see why/how to use ServletConfig interface in java servlet….

* ServletConfig is one of the pre-defined **interface**.
* ServletConfig object is used for developing **flexible servlets**.
* ServletConfig **objct exist one per servlet program**.
* An object of ServletConfig created by the **container during its initialization** phase.
* An object of ServletConfig is available to the servlet **during its execution**, once the servlet execution is completed, automatically ServletConfig interface object will be removed by the container.
* An object of ServletConfig interface **contains <init-param> details at web.xml**, of a particular servlet.
* The moment when we are **using an object of ServletConfig**, we **need to configure the web.xml** by writing **<init-param>** tag **under <servlet> tag of web.xml.**
* When ever compiler executes **init() mehod then the ServletConfig** will be created in general.
* An object of **ServletConfig contain the <init-param> data in the form of key,value pairs, here the keys represents init param names and values are its values**, which are represented in the web.xml file

### How to Get ServletConfig Object into Servelt

An object of ServletConfig can be obtained in **2** ways, they are…

***Way 1***

ServletConfig conf = getServletConfig();

In the above statement, we are directly calling getServletConfig() method as it is available in **Servlet interface**, inherited into GenericServlet and defined and further inherited into HttpServlet and later inherited into our own servlet class.

***Way 2***

ServletConfig object will be available in init() method of the servlet.

  public void init(ServletConfig config)  
{  
// …………………  
}

So finally we are able to create ServletConfig object in our servlet class, then how to get the data from that object… ?

### How to Retrieve Data from ServletConfig Interface Object

In order to retrieve the data of the ServletConfig we have two methods, which are present in ServletConfig interface..

* public String getInitParameter(“param name”);
* public Enumeration getInitParameterNames();

I am not going to explain about these methods, these are similar to ‘[Retrieve Client Input Data in Servlet](https://www.java4s.com/java-servlet-tutorials/how-to-retrieve-client-input-data-in-servlet/)‘ but here we are retrieving values from web.xml that’s it.

### Example of ServletConfig

### Files Required

* index.html
* OnServletConfig.java
* web.xml

### index.html

123456<font face="verdana" size="2px">

<form action="onSCG" method="post">

Example on ServletConfig<br>

<input type="submit" value="Calculate Sum">

</form>

</font>

### OnServletConfig.java

12345678910111213141516171819202122232425262728package java4s;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletConfig;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class OnServletConfig extends HttpServlet

{

protected void doPost(HttpServletRequest req,HttpServletResponse res)throws ServletException,IOException

{

PrintWriter pw=res.getWriter();

res.setContentType("text/html");

ServletConfig conf=getServletConfig();

String s1=conf.getInitParameter("n1");

String s2=conf.getInitParameter("n2");

pw.println("n1 value is " +s1+ " and n2 is " +s2);

pw.close();

}

}

### web.xml

<web-app>

<servlet>

<servlet-name>onServletConfig</servlet-name>

<servlet-class>java4s.OnServletConfig</servlet-class>

<init-param>

<param-name> n1 </param-name>

<param-value> 100 </param-value>

</init-param>

<init-param>

<param-name> n2 </param-name>

<param-value> 200 </param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>onServletConfig</servlet-name>

<url-pattern>/onSCG</url-pattern>

</servlet-mapping>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

</welcome-file-list>

</web-app>

### Output



## Example of ServletContext in Java

[Servlets](https://www.java4s.com/java-servlet-tutorials/) » on Feb 2, 2013 [**{ 14 Comments }**](https://www.java4s.com/java-servlet-tutorials/example-of-servletcontext-in-java/#comments) By Sivateja

ServletContext is one of pre-defined interface available in javax.servlet.\*; Object of ServletContext interface is available one per web application. An object of ServletContext is automatically created by the container when the web application is deployed.

Assume there exist a web application with 2 servlet classes, and they need to get some technical values from web.xml, in this case ServletContext concept will works great, i mean **all servlets in the current web application can access these context values from the web.xml but its not the case in ServletConfig, there only particular servelet can access the values from the web.xml which were written under <servlet> tag,** hope you remember.  Have doubt ? just check [***Example of ServletConfig***](https://www.java4s.com/java-servlet-tutorials/example-of-servletconfig-in-java-servlet/)

### How to Get ServletContext Object into Our Servlet Class

In servlet programming we have 3 approaches for obtaining an object of ServletContextinterface

***Way 1***

ServletConfig conf= getServletConfig();  
ServletContext context = conf.getServletContext();

* First obtain an object of **ServletConfig** interface
* ServletConfig interface contain direct method to get Context object, getServletContext();

***Way 2***

Direct approach, just call getServletContext() method available in GenericServlet [pre-defined].  In general we are extending our class with HttpServlet, but we know HttpServlet is the sub class of GenericServlet.

public class Java4s extends HttpServlet  
{  
public void doGet/doPost(-,-)  
{  
//….  
}  
**ServletContext ctx = getServletContext();**  
}

***Way 3***

We can get the object of ServletContext by making use of HttpServletRequest object, we have direct method in HttpServletRequest interface.

public class Java4s extends HttpServlet  
{  
public void doGet/doPost(HttpServletRequest req,-)  
{  
ServletContext ctx = req.getServletContext();  
}  
}

### How to Retrieve Data from ServletConfig Interface Object

ServletContext provide these 2 methods, In order to retrieve the data from the web..xml [ In web.xml we have **write <context-param>tag to provide** the values, and this <context-param> should write outside of <servlet> tag as context should be accessed by all servlet classes ].

In general **database related properties will be written in this type of situation**, where every servlet should access the same data.

* public String getInitParameter(“param name”);
* public Enumeration getInitParameterNames();

I am not going to explain about these methods, these are similar to ‘[Retrieve Client Input Data in Servlet](https://www.java4s.com/java-servlet-tutorials/how-to-retrieve-client-input-data-in-servlet/)‘ but here we are retrieving values from web.xml that’s it.

### Example of ServletContext

### Files Required

* index.html
* OnServletContext.java
* web.xml

### index.html

123456<font face="verdana" size="2px">

<form action="onContext" method="post">

Example on ServletContext<br>

<input type="submit" value="Click Here">

</form>

</font>

### OnServletContext.java

package java4s;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletContext;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class OnServletContext extends HttpServlet

{

    protected void doPost(HttpServletRequest req,HttpServletResponse res)throws ServletException,IOException

    {

        PrintWriter pw=res.getWriter();

        res.setContentType("text/html");

        // I am using 2nd way to create Context object

        ServletContext context=getServletContext();

        String s1=context.getInitParameter("n1");

        String s2=context.getInitParameter("n2");

        pw.println("n1 value is " +s1+ " and n2 is " +s2);

       pw.close();

    }

}

### web.xml

<web-app>

    <context-param>

        <param-name> n1 </param-name>

        <param-value> 100 </param-value>

    </context-param>

   <context-param>

        <param-name> n2  </param-name>

       <param-value> 200 </param-value>

    </context-param>

    <servlet>

        <servlet-name>onServletContext</servlet-name>

        <servlet-class>java4s.OnServletContext</servlet-class>

    </servlet>

    <servlet-mapping>

            <servlet-name>onServletContext</servlet-name>

            <url-pattern>/onContext</url-pattern>

    </servlet-mapping>

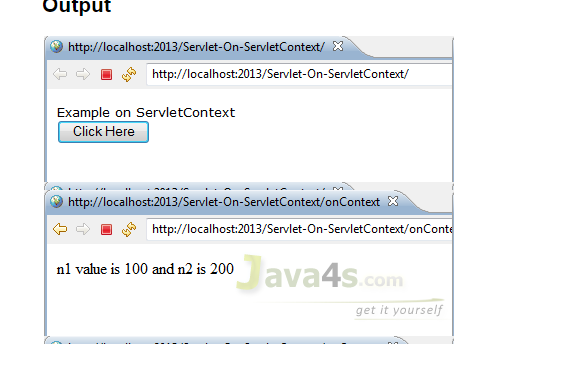
   <welcome-file-list>

        <welcome-file>index.html</welcome-file>

   </welcome-file-list>

</web-app>

### Output



## Difference between ServletConfig and ServletContext in Java

[Servlets](https://www.java4s.com/java-servlet-tutorials/) » on Feb 2, 2013 [**{ 39 Comments }**](https://www.java4s.com/java-servlet-tutorials/difference-between-servletconfig-and-servletcontext-in-java/#comments) By Sivateja

Let us see the main differences between ServletConfig and ServletContext, which is very popular interview question as well 🙂

### ServletConfig

* ServletConfig available in **javax**.**servlet**.\*; package
* ServletConfig object is **one** per servlet class
* Object of ServletConfig will be created during **initialization** process of the servlet
* This Config object is **public** to a **particular servlet** only
* **Scope**: As long as a servlet is executing, ServletConfig object will be available, it will be destroyed **once the servlet execution is completed.**
* We should give request explicitly, in order to create ServletConfig object for the first time
* In web.xml – **<init-param> tag will be appear under <servlet-class> tag**

### ServletContext

* ServletContext available in javax.servlet.\*; package
* ServletContext object is **global** to entire web application
* Object of ServletContext will be created at the time of web application **deployment**
* Scope: As long as web application is executing, ServletContext object will be available, and it will be destroyed once the **application is removed from the server**.
* **ServletContext object will be available even before giving the first request**
* In web.xml – <context-param> tag will be appear under **<web-app>** tag

So finally…….

No. of web applications  =  That many number of ServletContext objects [ 1 per web application ]  
No. of servlet classes = That many number of ServletConfig objects

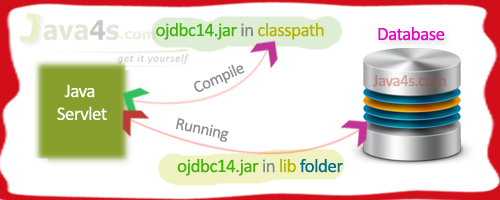
Cheers 😉

## How to Connect Servlet to the Database with Example

[Servlets](https://www.java4s.com/java-servlet-tutorials/) » on Feb 7, 2013 [**{ 18 Comments }**](https://www.java4s.com/java-servlet-tutorials/how-to-connect-servlet-to-the-database-with-example/#comments) By Sivateja

Let us see how to connect servelet application with (Oracle) database, for time being i am considering Oracle XE.  In our application i am going to display all the records from the table ‘Java4s‘.

* Make sure you have java4s table in the database with some data in it
* Make sure you have ojdbc14.jar file in your classpath and lib folder as well as [ why in two places ? 🙂 servlet will use ojdbc14.jar file which is in the classpath, for compiling the application, and ojdbc14.jar file in lib folder will be used at the time of running the servlet application], you can check the same in this figure…

[](https://www.java4s.com/wp-content/uploads/2013/02/servlet-database-connection-example.png)

### Directory Structure

|  |  |
| --- | --- |
|  |  |
|  |  |

### Files Required

* index.html
* ServletDatabaseConnect.java
* web.xml
* ojdbc14.jar

### index.html

<form action="show" method="post">

<font face="verdana" size="2">

Enter Table Name :<input type="text" name="table">

<input type="submit" value="Display">

</font>

</form>

### ServletDatabaseConnect.java

package java4s;

import java.io.IOException;

import java.io.PrintWriter;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.Statement;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class ServletDatabaseConnect extends HttpServlet

{

    protected void doPost(HttpServletRequest req,HttpServletResponse res)throws ServletException,IOException

    {

        PrintWriter pw=res.getWriter();

        res.setContentType("text/html");

        String tb=req.getParameter("table");

        try

        {

Class.forName("oracle.jdbc.driver.OracleDriver");

             Connection con=DriverManager.getConnection("jdbc:oracle:thin:*@localhost:1521:XE"*,"system","admin");

             Statement st=con.createStatement();

             System.out.println("connection established successfully...!!");

             ResultSet rs=**st.executeQuery**("Select \* from "+tb);

             pw.println("<table border=1>");

                 while(rs.next())

                 {

                     pw.println("<tr><td>"+rs.getInt(1)+"</td><td>"+rs.getString(2)+"</td>"+

                                      "<td>"+rs.getString(3)+"</td></tr>");

                 }

             pw.println("</table>");

             pw.close();

        }

        catch (Exception e){

            e.printStackTrace();

        }

    }

}

### web.xml

\<web-app>

    <servlet>

        <servlet-name>ServletDBConnect</servlet-name>

        <servlet-class>java4s.ServletDatabaseConnect</servlet-class>

    </servlet>

    <servlet-mapping>

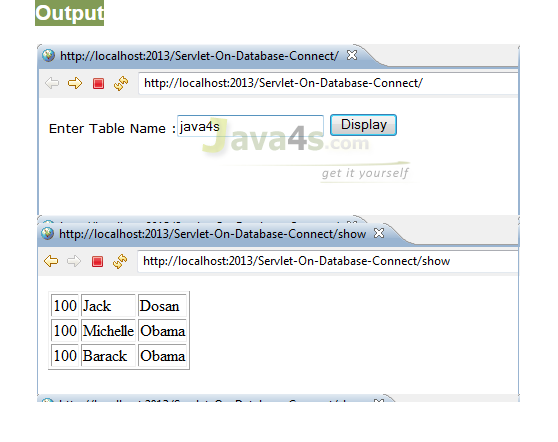
            <servlet-name>ServletDBConnect</servlet-name>

            <url-pattern>/show</url-pattern>

    </servlet-mapping>

</web-app>

### Output



# J2EE

## Difference between Java and J2EE

<http://www.differencebetween.net/technology/difference-between-java-and-j2ee/>

### Terminology of Java and J2EE

Java is a term given by Sun Microsystems to refer to the Java Standard Edition (Java SE). It’s a widely used programming language which derives much of its syntax from C and C++ with fewer implementation dependencies. J2EE, originally known as **Java Enterprise Edition (Java EE),** is a **collection of Java APIs** owned by Oracle Corporation used to write **server-side enterprise applications**.

### Platform for Java and J2EE

Java is a high-level class-based programming language that is commonly used to develop and deliver **content on the web**. It’s a **simplified version of C++** designed to run on all JVM-compliant platforms **irrespective of the architecture**. J2EE, on the other hand, is one of the core technologies of Java used for **developing server applications such as websites and web applications.**

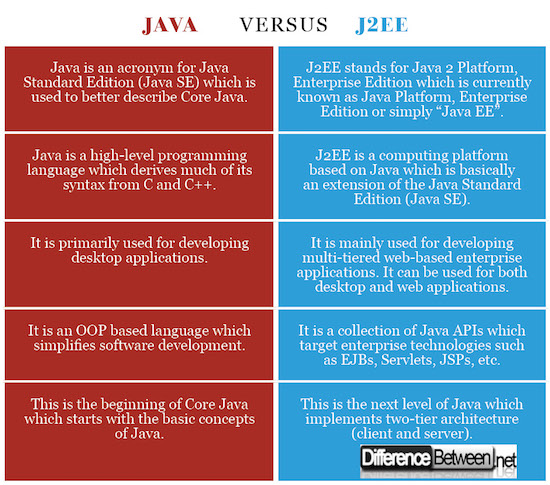
### Application of Java and J2EE

Java is robust and platform-independent, which means the compiled code can be executed on any platform running a Java Virtual Machine (JVM). It is write once and run anywhere language because of the bytecode which can run on different operating systems, making it a preferred choice for all kinds of platforms such as websites, servers, mobile phones, embedded systems, etc. J2EE, on the other hand, provides an architecture-neutral environment **to develop large-scale enterprise applications**.

### Components in Java and J2EE

The three main components of Java language are **Java Virtual Machine (JVM), Java Development Kit (JDK), and Java Runtime Environment (JRE)**. Each component works in conjunction with each other to simplify software development. J2EE, on the other hand, holds **Servlets** and **JavaServer Pages (JSPs),** along with **(Enterprise JavaBeans EJBs)** and **Java Database Connectivity (JDBC).**

### Java vs. J2EE: Comparison Chart



## Summary of Java vs. J2EE

Java is a general purpose programming language which is somewhat related to C++ which is in fact a direct descendant of the C language. Because much of the syntax of Java is inherited from C and C++, it would be wise to think of Java as an internet version of the C++. However, both share significant differences so comparing the two wouldn’t be a great idea either. Java wasn’t designed to replace C++, but to overcome the complexities of C++. It’s developed by Sun Microsystems which was later acquired by Oracle Corporation. Java Platform Enterprise Edition (Java EE), formerly known as J2EE, on other hand, is a standard for developing web-based server side applications online. **It is a collection of Java APIs owned by Oracle with a powerful set of libraries that can be used to build enterprise applications**.

Read more: [Difference Between Java and J2EE | Difference Between](http://www.differencebetween.net/technology/difference-between-java-and-j2ee/#ixzz65zPwCZoc) <http://www.differencebetween.net/technology/difference-between-java-and-j2ee/#ixzz65zPwCZoc>

# Multithreading in Java

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.  
  
Threads can be created by using two mechanisms :  
1. Extending the Thread class  
2. Implementing the Runnable Interface

**Thread creation by extending the Thread class**  
  
We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

|  |
| --- |
| // Java code for thread creation by extending  // the Thread class  class MultithreadingDemo extends Thread  {      public void run()      {          try          {              // Displaying the thread that is running              System.out.println ("Thread " +                    Thread.currentThread().getId() +                    " is running");            }          catch (Exception e)          {              // Throwing an exception              System.out.println ("Exception is caught");          }      }  }    // Main Class  public class Multithread  {      public static void main(String[] args)      {          int n = 8; // Number of threads          for (int i=0; i<8; i++)          {              MultithreadingDemo object = new MultithreadingDemo();              object.start();          }      }  } |

**Thread creation by implementing the Runnable Interface**  
  
We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

|  |
| --- |
| // Java code for thread creation by implementing  // the Runnable Interface  class MultithreadingDemo implements Runnable  {      public void run()      {          try          {              // Displaying the thread that is running              System.out.println ("Thread " +                                  Thread.currentThread().getId() +                                  " is running");            }          catch (Exception e)          {              // Throwing an exception              System.out.println ("Exception is caught");          }      }  }    // Main Class  class Multithread  {      public static void main(String[] args)      {          int n = 8; // Number of threads          for (int i=0; i<8; i++)          {              Thread object = new Thread(new MultithreadingDemo());              object.start();          }      }  } |

**Thread Class vs Runnable Interface**  
  
1. If we extend the Thread class, our class cannot extend any other class because Java doesn’t support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.  
  
2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

## Thread.sleep in Java

Thread.sleep() method can be used to pause the execution of current thread for specified time in milliseconds. The argument value for milliseconds can’t be negative, else it throws IllegalArgumentException.

There is another overloaded method sleep(long millis, int nanos) that can be used to pause the execution of current thread for specified milliseconds and nanoseconds. The allowed nano second value is between 0 and 999999.

## Java Thread Sleep important points

1. It always pause the current thread execution.
2. The actual time thread sleeps before waking up and start execution depends on system timers and schedulers. For a quiet system, the actual time for sleep is near to the specified sleep time but for a busy system it will be little bit more.
3. Thread sleep doesn’t lose any monitors or locks current thread has acquired.
4. Any other thread can interrupt the current thread in sleep, in that case InterruptedException is thrown.

## How Thread Sleep Works

Thread.sleep() interacts with the thread scheduler to put the current thread in wait state for specified period of time. Once the wait time is over, thread state is changed to runnable state and wait for the CPU for further execution. So the actual time that current thread sleep depends on the thread scheduler that is part of operating system.

# Collections

## Difference between Traditional Collections and Concurrent Collections in java

We all know about about Traditional Collections ( i.e. [List](https://www.geeksforgeeks.org/list-interface-java-examples/), [Set](https://www.geeksforgeeks.org/set-in-java/), [Queue](https://www.geeksforgeeks.org/queue-interface-java/) and its implemented Classes) and Concurrent Collection (i.e. ConcurrentMap interface, ConcurrentHashMap class, CopyOnWriteArrayList class etc). In these two Collections, there are few differences like:

* Most of the Classes which are present in **Traditional Collections (i.e**[**ArrayList**](https://www.geeksforgeeks.org/arraylist-in-java/)**,**[**LinkedList**](https://www.geeksforgeeks.org/linked-list-in-java/)**,**[**HashMap**](https://www.geeksforgeeks.org/hashmap-treemap-java/)**etc)** are non-synchronized in nature and Hence there is no thread-safety. But All the classes present in Concurrent Collections are synchronized in nature. Therefore In Concurrent classes, we dont have to take care about Thread-safety.
* While Traditional Collections also have **some classes (like**[**Vector**](https://www.geeksforgeeks.org/java-util-vector-class-java/)**,**[**Stack**](https://www.geeksforgeeks.org/stack-class-in-java/)**etc)** which are synchronized in nature and Traditional Collections also have **SynchronizedSet, SynchronizedList, SynchronizedMap** methods through which we can get Synchronized version of non-synchronized objects. But these above Synchronized classes are not good in terms of performance because of wide-locking mechanism .Whereas Concurrent Collections classes performance are relatively high than Traditional Collections classes.
* In the Traditional Collections, if a thread is iterating a Collection object and if another thread try to add new element in that iterating object simultaneously then we will get **RuntimeException ConcurrentModificationException**. Whereas In the above case, we will not get any Runtime Exception if we are Working with Concurrent Collections Classes.
* Traditional Collections classes is good choice if we are not dealing with thread in our application. whereas because of the Concurrent/Synchronized Collection we can use multiple Threads which are dealing with Collections Object. Therefore Concurrent Collections are best choice if we are dealing Multiple Threads in our application.

|  |
| --- |
| // Java program to illustrate Traditional  // Collections Problem  import java.util.\*;  class ConcurrentDemo extends Thread {      static ArrayList l = new ArrayList();      public void run()      {          try {              Thread.sleep(2000);          }          catch (InterruptedException e) {              System.out.println("Child Thread"                      + " going to add element");          }            // Child thread trying to add new          // element in the Collection object          l.add("D");      }        public static void main(String[] args)          throws InterruptedException      {          l.add("A");          l.add("B");          l.add("c");            // We create a child thread that is          // going to modify ArrayList l.          ConcurrentDemo t = new ConcurrentDemo();          t.start();            // Now we iterate through the ArrayList          // and get exception.          Iterator itr = l.iterator();          while (itr.hasNext()) {              String s = (String)itr.next();              System.out.println(s);              Thread.sleep(6000);          }          System.out.println(l);      }  } |

Output:

Exception in thread “main” java.util.ConcurrentModificationException

|  |
| --- |
| // Java program to illustrate ConcurrentCollection uses  import java.util.concurrent.CopyOnWriteArrayList;  import java.util.\*;  class ConcurrentDemo extends Thread {      static CopyOnWriteArrayList l =                       new CopyOnWriteArrayList();      public void run()      {          try {              Thread.sleep(2000);          }          catch (InterruptedException e) {              System.out.println("Child Thread"                       + " going to add element");          }            // Child thread trying to add new          // element in the Collection object          l.add("D");      }        public static void main(String[] args)          throws InterruptedException      {          l.add("A");          l.add("B");          l.add("c");            // We create a child thread that is          // going to modify ArrayList l.          ConcurrentDemo t = new ConcurrentDemo();          t.start();            // Now we iterate through the ArrayList          // and get exception.          Iterator itr = l.iterator();          while (itr.hasNext()) {              String s = (String)itr.next();              System.out.println(s);              Thread.sleep(6000);          }          System.out.println(l);      }  } |

output:

A

B

c

## Difference Between Arraylist And Vector : Core Java Interview Collection Question

### What is bucket ? A bucket is used to store key value pairs . A bucket can have multiple key-value pairs . In hash map, bucket used simple linked list to store objects .

## Constructors for Java LinkedList:

1. LinkedList(): Used to create an empty linked list.
2. LinkedList(Collection C): Used to create a ordered list which contains all the elements of a specified collection, as returned by the collection’s iterator.

## How To Add Element At Specified Index In ArrayList : Add(Int Index, E Element) Example

add(index, element) which **adds element at the specified index**.  
  
The syntax for the method is  
  
**public void add(int index, Object element)**

**mport** **java.util.\***;

**public** **class** **AddMethodExample** {

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

 // Declaration of String ArrayList

ArrayList<String> al = **new** ArrayList<String>();

/\* Simple add() method for adding element

at the end of the ArrayList \*/

al.add("California");

al.add("Boston");

al.add("San jose");

al.add("New York");

//Adding element to the 3rd position

//3rd position = 2 index as index starts with 0

al.add(**2**,"San Francisco");

System.out.println("ArrayList after adding String San Francisco:"+ al);

//Addition of String element at 1st position

al.add(**0**, "Texas");

//Displaying the ArrayList

System.out.println("ArrayList after adding String Texas:"+ al);

}

}

**Output**

ArrayList after adding String San Francisco:[California, Boston, San Francisco, San jose, New York]

ArrayList after adding String Texas:[Texas, California, Boston, San Francisco, San jose, New York]

## Remove Element At The Specified Index In ArrayList: Remove(Int Index) Example

In the last tutorial I have shared the [addAll(int index, Collection c) method example](http://javahungry.blogspot.com/2017/10/how-to-insert-all-collection-elements-at-specified-position-arraylist.html). In this tutorial we will learn about remove(int index) method which is used for removing an element at the specified index from an ArrayList.You can learn about [how ArrayList add(Object) method works internally in java](http://javahungry.blogspot.com/2015/05/how-add-method-works-internally-in-arraylist.html). Syntax :  
  
public Object remove(int index)  
  
**remove(int index) example:**

**import** **java.util.\***;

**public** **class** **RemoveMethodExample** {

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

//String ArrayList

ArrayList<String> al = **new** ArrayList<String>();

al.add("AA");

al.add("BB");

al.add("CC");

al.add("DD");

al.add("AA");

al.add("ZZ");

System.out.println("ArrayList before remove:");

**for**(String **var:** al){

System.out.println(var);

}

//Removing 1st element

al.remove(**0**);

//Removing 3rd element from the remaining list

al.remove(**2**);

//Removing 4th element from the remaining list

al.remove(**2**);

System.out.println("ArrayList After remove:");

**for**(String **var:** al){

System.out.println(var);

}

}

}

## Constructors in Java ArrayList:

1. ArrayList(): This constructor is used to build an empty array list
2. ArrayList(Collection c): This constructor is used to build an array list initialized with the elements from collection c // used for conversion of any list to ArrayList **but cant convert array to arraylist by this way as array is not collection.**
3. ArrayList(int capacity): This constructor is used to build an array list with initial capacity being specified

### Synchronization and Thread-Safe Vector is  synchronized while ArrayList is not synchronized  . Synchronization and thread safe means at a time only one thread can access the code .In Vector class all the methods are synchronized .Thats why the Vector object is already synchronized when it is created . 2.  Performance Vector is slow as it is thread safe . In comparison ArrayList is fast as it is non synchronized . Thus     in ArrayList two or more threads  can access the code at the same time  , while Vector is limited to one thread at a time. 3. Automatic Increase in Capacity A Vector defaults to doubling size of its array . While when you insert an element into the ArrayList ,      it increases its Array size by 50%  . By default ArrayList size is 10 . It checks whether it reaches the       last  element then it will create the new array ,copy the new data of last array to new array ,then old array     is garbage collected by the Java Virtual Machine (JVM) . 4. Set Increment Size ArrayList does not define the increment size . Vector defines the increment size . You can find the following method in Vector Class public synchronized void setSize(int i) { //some code  } There is no setSize() method or any other method in ArrayList which can manually set the increment size. 5. Enumerator Other than Hashtable ,Vector is the only other class which uses both [**Enumeration and Iterator**](https://javahungry.blogspot.com/2013/06/difference-between-iterator-and-enumeration-collections-java-interview-question-with-example.html) .While ArrayList can only use Iterator for traversing an ArrayList . 6.  Introduction in Java java.util.Vector  class was there in java since the very first version of the java development kit (jdk). java.util.ArrayList  was introduced in java version 1.2 , as part of Java Collections framework . In java version 1.2 , Vector class has been refactored to implement the List Inteface .

## Constructors in HashSet:

1. **HashSet h = new HashSet();**  
   Default initial capacity is 16 and default load factor is 0.75.
2. **HashSet h = new HashSet(int initialCapacity);**  
   default loadFactor of 0.75
3. **HashSet h = new HashSet(int initialCapacity, float loadFactor**);
4. **HashSet h = new HashSet(Collection C);**

## HashMap containsKey() Method in Java

The java.util.HashMap.containsKey() method is used to check whether a particular key is being mapped into the HashMap or not. It takes the key element as a parameter and returns True if that element is mapped in the map.

**Syntax:**

Hash\_MapKey(*key\_element*)

**Parameters:** The method takes just one parameter *key\_element* that refers to the key whose mapping is supposed to be checked inside a map.

**Return Value:** The method returns boolean true if the presence of the key is detected else false .

## Convert HashSet To Array In Java With Example

We will use toArray() method to convert HashSet to an Array.  
  
  
**Program to Convert HashSet to Array in Java**

1. **import** **java.util.\***;
2. **public** **class** **HashSettoArray** {
3. **public** **static** **void** **main**(String args[]) {
5. // Declaring a HashSet
6. HashSet<String> hashset = **new** HashSet<String>();
7. // Add elements to HashSet
8. hashset.add("Pear");
9. hashset.add("Apple");
10. hashset.add("Orange");
11. hashset.add("Papaya");
12. hashset.add("Banana");
13. //Showing HashSet elements
14. System.out.println("HashSet contains :" + hashset);
15. // Creating an Array of HashSet size
16. String[] array = **new** String[hashset.size()];
17. // Converting HashSet to Array using toArray() method
18. hashset.toArray(array);
19. //Showing Array elements
20. System.out.println("Array contains :");
21. **for**(String str : array){
22. System.out.println(str);
23. }
24. }
25. }
26. **Output**
27. HashSet contains :[Apple, Pear, Papaya, Orange, Banana]
28. Array contains :
29. Apple
30. Pear
31. Papaya
32. Orange
33. Banana

## How To Convert LinkedList To ArrayList In Java

1. I have already shared what is the [difference between LinkedList and ArrayList in java](http://javahungry.blogspot.com/2015/04/difference-between-arraylist-and-linkedlist-in-java-example.html). In this tutorial we are converting a LinkedList to ArrayList in java. We have a LinkedList of Strings in which we are storing name of the fruits. Later after conversion we are showing the elements of ArrayList to ensure that ArrayList is having the same elements that we have in LinkedList.  
     
   According to [Oracle docs,](https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html) we are using ArrayList constructor to convert LinkedList to ArrayList. Syntax is  
     
   new ArrayList(Collection c)  
     
   where we pass LinkedList to the Collection c and it will convert to the ArrayList.  
     
   **Convert LinkedList to ArrayList Example**
2. **import** **java.util.\***;
3. **class** **ConvertExample** {
4. **public** **static** **void** **main**(String args[]) {
5. // Creating LinkedList Object
6. LinkedList<String> linkedlist = **new** LinkedList<String>();
7. linkedlist.add("Mango");
8. linkedlist.add("Banana");
9. linkedlist.add("Pear");
10. linkedlist.add("Apple");
11. linkedlist.add("Orange");
12. // Converting LinkedList to ArrayList
13. List<String> list = **new** ArrayList<String>(linkedlist);
14. **for** (String str : list){
15. System.out.println(str);
16. }
17. }
18. }

## Conversion of Array To ArrayList in Java

We can convert an array to arraylist using following ways.

1. **Using Arrays.asList() method** - Pass the array and get a arraylist.
2. **Collections.addAll() method** - Create a new list before using this method and then add array elements using this method to existing list.
3. **Iteration method** - Create a new list. Iterate the array and add each element to the list.

#### Example

[Live Demo](http://tpcg.io/aUCZnx)

import java.util.ArrayList;

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public class Tester {

   public static void main(String args[]) {

      String[] array = {"a", "b", "c", "d", "e"};

      //Method 1

      List<String> list = Arrays.asList(array);

      System.out.println(list);

      //Method 2

      List<String> list1 = new ArrayList<String>();

      Collections.addAll(list1, array);

      System.out.println(list1);

      //Method 3

      List<String> list2 = new ArrayList<String>();

      for(String text:array) {

         list2.add(text);

      }

      System.out.println(list2);

   }

}

#### Output

[a, b, c, d, e]

[a, b, c, d, e]

[a, b, c, d, e]

## Program to convert Array to List in Java

**Examples:**

***Input:****Array: [Geeks, forGeeks, A computer Portal]****Output:****List: [Geeks, forGeeks, A computer Portal]*

***Input:****Array: [1, 2, 3, 4, 5]****Output:****List: [1, 2, 3, 4, 5]*

Below are methods to convert Array to List in Java:

1. **Brute Force or Naive Method**: In this method, an empty List is created and all elements present of the Array are added to it one by one.

**Algorithm**:

* 1. Get the Array to be converted.
  2. Create an empty List
  3. Iterate through the items in the Array.
  4. For each item, add it to the List
  5. Return the formed List

**Program:**

|  |
| --- |
| // Java Program to convert  // Array to List    import java.util.\*;  import java.util.stream.\*;    class GFG {        // Generic function to convert an Array to List      public static <T> List<T> convertArrayToList(T array[])      {            // Create an empty List          List<T> list = new ArrayList<>();            // Iterate through the array          for (T t : array) {              // Add each element into the list              list.add(t);          }            // Return the converted List          return list;      }        public static void main(String args[])      {          // Create an Array          String array[] = { "Geeks", "forGeeks",                                      "A Computer Portal" };            // Print the Array          System.out.println("Array: "                         + Arrays.toString(array));            // convert the Array to List          List<String>              list = convertArrayToList(array);            // Print the List          System.out.println("List: " + list);      }  } |

**Output:**

Array: [Geeks, forGeeks, A computer Portal]

List: [Geeks, forGeeks, A computer Portal]

1. **Using Arrays.asList() method**: In this method, the Array is passed as the parameter into the List constructor with the help of Arrays.asList() method.

**Algorithm**:

* 1. Get the Array to be converted.
  2. Create the List by passing the Array as parameter in the constructor of the List with the help of Arrays.asList() method
  3. Return the formed List

**Program:**

// Java Program to convert

2

// Array to List 4

import java.util.\*;

5

import java.util.stream.\*; 7

class GFG {

9

// Generic function to convert an Array to List

10

public static <T> List<T> convertArrayToList(T array[])

11

{

13

// Create the List by passing the Array

14

// as parameter in the constructor

15

List<T> list = Arrays.asList(array);

16

​

17

// Return the converted List

18

return list;

19

}

21

public static void main(String args[])

22

{

23

// Create an Array

24

String array[] = { "Geeks", "forGeeks",

25

"A computer Portal" };

27

// Print the Array

28

System.out.println("Array: "

29

+ array);

30

// System.out.println("Array: "

31

// + Arrays.toString(array));

32

​

33

// convert the Array to List

34

List<String>

35

list = convertArrayToList(array);

36

​

37

// Print the List

38

System.out.println("List: " + list);

39

}

40

}

41

​

**chevron\_right**

Array: [Ljava.lang.String;@232204a1

List: [Geeks, forGeeks, A computer Portal]

**Output:**

Array: [Geeks, forGeeks, A computer Portal]

List: [Geeks, forGeeks, A computer Portal]

1. **Using Collections.addAll()**: Since List is a part of the Collection package in Java. Therefore the Array can be converted into the List with the help of Collections.addAll() method.

**Algorithm**:

* 1. Get the Array to be converted.
  2. Create an empty List.
  3. Add the array into the List by passing it as the parameter to the Collections.addAll() method.
  4. Return the formed List

**Program:**

|  |
| --- |
| // Java Program to convert  // Array to List    import java.util.\*;  import java.util.stream.\*;    class GFG {        // Generic function to convert an Array to List      public static <T> List<T> convertArrayToList(T array[])      {            // Create the List by passing the Array          // as parameter in the constructor          List<T> list = new ArrayList<>();            // Add the array to list          Collections.addAll(list, array);            // Return the converted List          return list;      }        public static void main(String args[])      {          // Create an Array          String array[] = { "Geeks", "forGeeks",                                      "A computer Portal" };            // Print the Array          System.out.println("Array: "                                + Arrays.toString(array));            // convert the Array to List          List<String>              list = convertArrayToList(array);            // Print the List          System.out.println("List: " + list);      }  } |

**Output:**

Array: [Geeks, forGeeks, A computer Portal]

List: [Geeks, forGeeks, A computer Portal]

**Output:**

Array: [Geeks, forGeeks, A computer Portal]

List: [Geeks, forGeeks, A computer Portal]

## How To Convert ArrayList to String Array In Java

[Core Java](https://www.java4s.com/core-java/) » on Sep 27, 2012 [**{ 24 Comments }**](https://www.java4s.com/core-java/how-to-convert-arraylist-to-string-array-in-java/#comments) By Sivateja

Hi friends,  In java some times we may need some conversions from collection objects to some thing, ArrayList to String array is very popular conversion we are using in our projects 🙂

1234567891011121314151617181920212223import java.util.ArrayList;

import java.util.List;

public class Java4s {

public static void main(String args[]){

List al = new ArrayList<String>();

al.add("One");

al.add("Two");

al.add("Three");

al.add("Four");

al.add("Five");

String[] stringArrayObject = new String[al.size()];

al.toArray(stringArrayObject);

for(String temp : stringArrayObject)

System.out.println(temp);

}

}

### Explanation

Line number 8 created ArrayList object  
Line numbers 10,11,12,13,14 added data into list object  
Line number 16 created String array, and at 17 we converted List data into String array

Finally output 😉

## keyset vs entryset

The **entrySet**() method is used to get a Set view of the mappings contained in this map. The **keySet**() method is used to get a Set view of the keys contained in this map.

# HashMap:

## How To Remove All Mappings From HashMap In Java With Example

We will use clear() method for this.

**import** **java.util.\***;

**public** **class** **HashMapRemoveExample** {

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

// Creating a HashMap of int keys and String values

HashMap<Integer, String> hashmap = **new** HashMap<Integer, String>();

// Adding Key and Value pairs to HashMap

hashmap.put(**11**,"Value1");

hashmap.put(**22**,"Value2");

hashmap.put(**33**,"Value3");

hashmap.put(**44**,"Value4");

hashmap.put(**55**,"Value5");

hashmap.put(**66**,"Value6");

// Showing HashMap Elements

System.out.println("HashMap Elements: " + hashmap);

// Removes all Mapping

hashmap.clear();

// Showing HashMap Elements after calling clear()

System.out.println("After calling clear():");

System.out.println("---------------------");

System.out.println("HashMap Elements: " + hashmap);

}

}

**Output**

HashMap Elements: {33=Value3, 66=Value6, 22=Value2, 55=Value5, 11=Value1, 44=Value4}

After calling clear():

## HashMap containsValue() Method in Java

The java.util.HashMap.containsValue() method is used to check whether a particular value is being mapped by a single or more than one key in the HashMap. It takes the Value as a parameter and returns True if that value is mapped by any of the key in the map.

**Syntax:**

Hash\_Map.containsValue(*Object Value*)

**Parameters:** The method takes just one parameter *Value* of Object type and refers to the value whose mapping is supposed to be checked by any key inside the map.

**Return Value:** The method returns boolean true if the mapping of the value is detected else false.

## HashMap entrySet() Method in Java

The java.util.HashMap.entrySet() method in Java is used to create a set out of the same elements contained in the hash map. It basically returns a set view of the hash map or we can create a new set and store the map elements into them.

**Syntax:**

hash\_map.entrySet()

**Parameters:** The method does not take any parameter.

**Return Value:** The method returns a set having same elements as the hash map.

Below programs are used to illustrate the working of java.util.HashMap.entrySet() Method:  
**Program 1:** Mapping String Values to Integer Keys.

|  |
| --- |
| // Java code to illustrate the entrySet() method  import java.util.\*;    public class Hash\_Map\_Demo {      public static void main(String[] args)      {            // Creating an empty HashMap          HashMap<Integer, String> hash\_map = new HashMap<Integer, String>();            // Mapping string values to int keys          hash\_map.put(10, "Geeks");          hash\_map.put(15, "4");          hash\_map.put(20, "Geeks");          hash\_map.put(25, "Welcomes");          hash\_map.put(30, "You");            // Displaying the HashMap          System.out.println("Initial Mappings are: " + hash\_map);            // Using entrySet() to get the set view          System.out.println("The set is: " + hash\_map.entrySet());      }  } |

**Output:**

Initial Mappings are: {20=Geeks, 25=Welcomes, 10=Geeks, 30=You, 15=4}

The set is: [20=Geeks, 25=Welcomes, 10=Geeks, 30=You, 15=4]

## HashMap putAll() Method in Java

The java.util.HashMap.putAll() is an inbuilt method of HashMap class that is used for the copy operation. The method copies all of the elements i.e., the mappings, from one map into another.

**Syntax:**

new\_hash\_map.putAll(*exist\_hash\_map*)

**Parameters:** The method takes one parameter *exist\_hash\_map* that refers to the existing map we want to copy from.

**Return Value:** The method does not return any values.

**Exception:** The method throws *NullPointerException* if the map we want to copy from is NULL.

Below programs illustrates the working of java.util.HashMap.putAll() method:  
**Program 1:** Mapping String Values to Integer Keys.

|  |
| --- |
| // Java code to illustrate the putAll() method  import java.util.\*;    public class Hash\_Map\_Demo {  public static void main(String[] args) {        // Creating an empty HashMap      HashMap<Integer, String> hash\_map = new HashMap<Integer, String>();        // Mapping string values to int keys      hash\_map.put(10, "Geeks");      hash\_map.put(15, "4");      hash\_map.put(20, "Geeks");      hash\_map.put(25, "Welcomes");      hash\_map.put(30, "You");        // Displaying the HashMap      System.out.println("Initial Mappings are: " + hash\_map);        // Creating a new hash map and copying      HashMap<Integer, String> new\_hash\_map = new HashMap<Integer, String>();      new\_hash\_map.putAll(hash\_map);        // Displaying the final HashMap      System.out.println("The new map looks like this: " + new\_hash\_map);  }  } |

**Output:**

Initial Mappings are: {20=Geeks, 25=Welcomes, 10=Geeks, 30=You, 15=4}

The new map looks like this: {25=Welcomes, 10=Geeks, 20=Geeks, 30=You, 15=4}

**Program 2:** Mapping Integer Values to String Keys.

|  |
| --- |
| // Java code to illustrate the putAll() method  import java.util.\*;    public class Hash\_Map\_Demo {      public static void main(String[] args)      {            // Creating an empty HashMap          HashMap<String, Integer> hash\_map = new HashMap<String, Integer>();            // Mapping int values to string keys          hash\_map.put("Geeks", 10);          hash\_map.put("4", 15);          hash\_map.put("Geeks", 20);          hash\_map.put("Welcomes", 25);          hash\_map.put("You", 30);            // Displaying the HashMap          System.out.println("Initial Mappings are: " + hash\_map);            // Creating a new hash map and copying          HashMap<String, Integer> new\_hash\_map = new HashMap<String, Integer>();          new\_hash\_map.putAll(hash\_map);            // Displaying the final HashMap          System.out.println("The new map looks like this: " + new\_hash\_map);      }  } |

**Output:**

Initial Mappings are: {4=15, Geeks=20, You=30, Welcomes=25}

The new map looks like this: {Geeks=20, 4=15, You=30, Welcomes=25}

## HashMap keySet() Method in Java

The java.util.HashMap.keySet() method in Java is used to create a set out of the key elements contained in the hash map. It basically returns a set view of the keys or we can create a new set and store the key elements in them.

**Syntax:**

hash\_map.keySet()

**Parameters:** The method does not take any parameter.

**Return Value:** The method returns a set having the keys of the hash map.

Below programs are used to illustrate the working of java.util.HashMap.keySet() Method:  
**Program 1:** Mapping String Values to Integer Keys.

|  |
| --- |
| // Java code to illustrate the keySet() method  import java.util.\*;    public class Hash\_Map\_Demo {      public static void main(String[] args)      {            // Creating an empty HashMap          HashMap<Integer, String> hash\_map = new HashMap<Integer, String>();            // Mapping string values to int keys          hash\_map.put(10, "Geeks");          hash\_map.put(15, "4");          hash\_map.put(20, "Geeks");          hash\_map.put(25, "Welcomes");          hash\_map.put(30, "You");            // Displaying the HashMap          System.out.println("Initial Mappings are: " + hash\_map);            // Using keySet() to get the set view of keys          System.out.println("The set is: " + hash\_map.keySet());      }  } |

**Output:**

Initial Mappings are: {20=Geeks, 25=Welcomes, 10=Geeks, 30=You, 15=4}

The set is: [20, 25, 10, 30, 15]

## Map.Entry interface in Java with example

Map.Entry interface in Java provides certain methods to access the entry in the Map. By gaining access to the entry of the Map we can easily manipulate them. Map.Entry is a generic and is defined in the java.util package.

## HashMap get() Method in Java

The java.util.HashMap.get() method of HashMap class is used to retrieve or fetch the value mapped by a particular key mentioned in the parameter. It returns NULL when the map contains no such mapping for the key.

**Syntax:**

Hash\_Map.get(*Object key\_element*)

**Parameter:** The method takes one parameter *key\_element* of object type and refers to the key whose associated value is supposed to be fetched.

**Return Value:** The method returns the value associated with the *key\_element* in the parameter.

Below programs illustrates the working of java.util.HashMap.get() method:  
**Program 1:** Mapping String Values to Integer Keys.

|  |
| --- |
| // Java code to illustrate the get() method  import java.util.\*;    public class Hash\_Map\_Demo {      public static void main(String[] args)      {            // Creating an empty HashMap          HashMap<Integer, String> hash\_map = new HashMap<Integer, String>();            // Mapping string values to int keys          hash\_map.put(10, "Geeks");          hash\_map.put(15, "4");          hash\_map.put(20, "Geeks");          hash\_map.put(25, "Welcomes");          hash\_map.put(30, "You");            // Displaying the HashMap          System.out.println("Initial Mappings are: " + hash\_map);            // Getting the value of 25          System.out.println("The Value is: " + hash\_map.get(25));            // Getting the value of 10          System.out.println("The Value is: " + hash\_map.get(10));      }  } |

**Output:**

Initial Mappings are: {20=Geeks, 25=Welcomes, 10=Geeks, 30=You, 15=4}

The Value is: Welcomes

The Value is: Geeks

**Program 2:** Mapping Integer Values to String Keys.

|  |
| --- |
| // Java code to illustrate the get() method  import java.util.\*;    public class Hash\_Map\_Demo {      public static void main(String[] args)      {            // Creating an empty HashMap          HashMap<String, Integer> hash\_map = new HashMap<String, Integer>();            // Mapping int values to string keys          hash\_map.put("Geeks", 10);          hash\_map.put("4", 15);          hash\_map.put("Geeks", 20);          hash\_map.put("Welcomes", 25);          hash\_map.put("You", 30);            // Displaying the HashMap          System.out.println("Initial Mappings are: " + hash\_map);            // Getting the value of "Geeks"          System.out.println("The Value is: " + hash\_map.get("Geeks"));            // Getting the value of "You"          System.out.println("The Value is: " + hash\_map.get("You"));      }  } |

**Output:**

Initial Mappings are: {4=15, Geeks=20, You=30, Welcomes=25}

The Value is: 20

The Value is: 30

**Note:** The same operation can be performed with any type of Mappings with variation and combination of different data types.

How To Sort HashMap In Java By Keys And Values

HashMap Sorting by Keys   
In this example we are sorting the HashMap based on the keys using TreeMap collection class.  
Program for Sorting HashMap by Keys

import java.util.\*;

public class HashMapSortByKeyExample {

public static void main(String args[]) {

// Creating a HashMap of int keys and String values

HashMap<Integer, String> hashmap = new HashMap<Integer, String>();

// Adding Key and Value pairs to HashMap

hashmap.put(22,"A");

hashmap.put(55,"B");

hashmap.put(33,"Z");

hashmap.put(44,"M");

hashmap.put(99,"I");

hashmap.put(88,"X");

System.out.println("Before Sorting:");

/\* Set set = hashmap.entrySet();

Iterator iterator = set.iterator();

while(iterator.hasNext()) {

Map.Entry pair = (Map.Entry)iterator.next();

System.out.print(pair.getKey() + ": ");

System.out.println(pair.getValue());

} \*/

for(Map.Entry<Integer,String> m:hashmap.entrySet())

{ System.out.println(m.getKey()+"="+m.getValue()); }

Map<Integer, String> map = new TreeMap<Integer, String>(hashmap);

System.out.println("After Sorting:");

/\* Set set2 = map.entrySet();

Iterator iterator2 = set2.iterator();

while(iterator2.hasNext()) {

Map.Entry pair = (Map.Entry)iterator2.next();

System.out.print(pair.getKey() + ": ");

System.out.println(pair.getValue());

}

\*/

for(Map.Entry<Integer,String> m:map.entrySet())

{ System.out.println(m.getKey()+"="+m.getValue()); }

}

}

HashMap Sorting by Values

 we are sorting HashMap by values using Comparator.

**Program for Sorting HashMap by Values**

**import** **java.util.\***;

**public** **class** **HashMapSortByValueExample** {

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

// Creating a HashMap of int keys and String values

HashMap<Integer, String> hashmap = **new** HashMap<Integer, String>();

// Adding Key and Value pairs to HashMap

hashmap.put(**22**,"A");

hashmap.put(**55**,"B");

hashmap.put(**33**,"Z");

hashmap.put(**44**,"M");

hashmap.put(**99**,"I");

hashmap.put(**88**,"X");

System.out.println("Before Sorting:");

Set set = hashmap.entrySet();

Iterator iterator = set.iterator();

**while**(iterator.hasNext()) {

Map.Entry pair = (Map.Entry)iterator.next();

System.out.print(pair.getKey() + ": ");

System.out.println(pair.getValue());

}

Map<Integer, String> map = sortByValues(hashmap);

System.out.println("After Sorting:");

Set set2 = map.entrySet();

Iterator iterator2 = set2.iterator();

**while**(iterator2.hasNext()) {

Map.Entry pair = (Map.Entry)iterator2.next();

System.out.print(pair.getKey() + ": ");

System.out.println(pair.getValue());

}

}

**private** **static** HashMap **sortByValues**(HashMap map) {

List list = **new** LinkedList(map.entrySet());

// Defined Custom Comparator here

Collections.sort(list, **new** Comparator() {

**public** **int** **compare**(Object o1, Object o2) {

**return** ((Comparable) ((Map.Entry) (o1)).getValue())

.compareTo(((Map.Entry) (o2)).getValue());

}

});

// Here I am copying the sorted list in HashMap

// using LinkedHashMap to preserve the insertion order

HashMap sortedHashMap = **new** LinkedHashMap();

**for** (Iterator it = list.iterator(); it.hasNext();) {

Map.Entry entry = (Map.Entry) it.next();

sortedHashMap.put(entry.getKey(), entry.getValue());

}

**return** sortedHashMap;

}

}

**Output**

Before Sorting:

33: Z

99: I

22: A

55: B

88: X

44: M

After Sorting:

22: A

55: B

99: I

44: M

88: X

33: Z

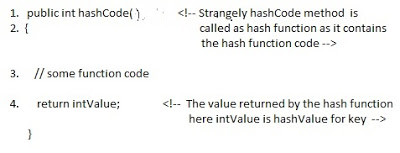
Hashing :How HashMap Works In Java Or How Get() Method Works Internally

<https://javahungry.blogspot.com/2013/08/hashing-how-hash-map-works-in-java-or.html>

Hashing :How HashMap Works In Java Or How Get() Method Works Internally

One of the most darling question of the core java interviewers is How HashMap works in java or internal implementation of HashMap. Most of the candidates rejection chances increases if the candidate do not give the satisfactory explanation . This question shows that candidate has good knowledge of Collection . So this question should be in your to do list before appearing for the interview. I have updated the article with deep explanation of java 8 changes, i.e, how HashMap works in java 8.  
  
Read also  [How Hashset works in java or How it ensures uniqueness in java](https://javahungry.blogspot.com/2013/08/how-sets-are-implemented-internally-in.html)  
                  [Java Interview questions for experienced](https://javahungry.blogspot.com/2013/06/top-25-most-frequently-asked-core-java.html)  
  
How HashMap works in Java  
  
HashMap works on the principle of Hashing .  To understand Hashing , we should understand the three terms first   i.e  *Hash Function , Hash Value and Bucket .*  
  
What is Hash Function , Hash Value  and Bucket ?  
  
hashCode() function  which returns an integer value is the Hash function. The important point to note that ,  this method is present in Object class ( Mother of all class ) .  
  
This is the code for the hash function(also known as hashCode method) in Object Class :  
  
    public native int hashCode();

The most important point to note from the above line :  hashCode method return  int value .  
So the Hash value is the int value returned by the hash function .  
  
  
    So summarize the terms in the diagram below :  
                  

[](https://3.bp.blogspot.com/-ohpWRYtP3N8/UgVGHwEk3YI/AAAAAAAAAbQ/K5BKPJ8dfmQ/s1600/How+hash+map+works+.jpg)

What is bucket ?  
A bucket is used to store key value pairs . A bucket can have multiple key-value pairs . In hash map, bucket used simple linked list to store objects .  
  
  
After understanding the terms we are ready to move next step , How HashMap works in java or How get() works internally in java .  
  
  
Code inside Java Api (HashMap class internal implementation) for HashMap get(Obejct key) method

1. Public V get(Object key)

{

2. if (key ==null)

3. //Some code

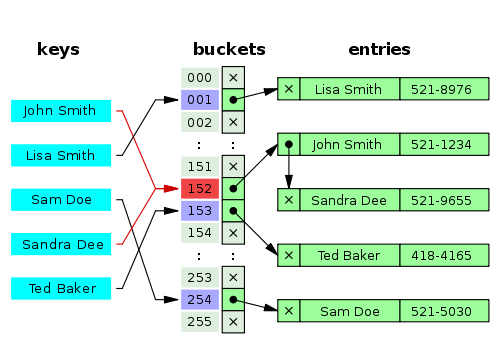
4. int hash = hash(key.hashCode());

5. // if key found in hash table then return value

6. // else return null

}

Hash map works on the principle of hashing   
  
HashMap get(Key k) method calls hashCode method on the key object and applies returned hashValue to its own static hash function to find a bucket location(backing array) where keys and values are stored in form of a nested class called Entry (Map.Entry) . So you have concluded that from the previous line that Both key and value is stored in the bucket as a form of  Entry object . So thinking that Only value is stored  in the bucket is not correct and will not give a good impression on the interviewer .  
  
\* Whenever we call get( Key k )  method on the HashMap object . First it checks that whether key is null or not .  Note that there can only be one null key in HashMap .    
  
If key is null , then Null keys always map to hash 0, thus index 0.  
  
If key is not null then , it will call hashfunction on the key object , see line 4 in above method i.e. key.hashCode()  ,so after key.hashCode() returns hashValue , line 4 looks like  
  
4.                int hash = hash(hashValue)  
  
 , and now ,it applies returned hashValue into its own hashing function .  
  
We might wonder why we are calculating the hashvalue again using hash(hashValue). Answer is ,It defends against poor quality hash functions.  
  
Now step 4 final  hashvalue is used to find the bucket location at which the Entry object is stored . Entry object stores in the bucket like this (hash,key,value,bucketindex) .    
  
Interviewer:    What if  when two different keys have the same hashcode ?  
  
Solution, [equals() method](http://javahungry.blogspot.com/2013/06/difference-between-equals-and-double-equals-method-with-example-java-collections-interview-question.html) comes to rescue.Here candidate gets puzzled. Since bucket is one and we have two objects with the same hashcode .Candidate usually forgets that bucket is a simple linked list.  
  
The bucket is the linked list effectively . Its not a LinkedList as in a java.util.LinkedList - It's a separate (simpler) implementation just for the map .  
  
So we traverse through linked list , comparing keys in each entries using keys.equals() until it return true.  Then the corresponding entry object Value is returned .

[](https://4.bp.blogspot.com/-unPwpp8AJTA/U0e9S0F5ljI/AAAAAAAAAUo/xMnUVRO5fyY/s1600/how+hashmap+works+internally+in+java+.png)

One of  our readers Jammy  asked a very good  question   
  
When the functions 'equals' traverses through the linked list does it traverses from start to end one by one...in other words brute method. Or the linked list is sorted based on key and then it traverses?  
  
Answer is when an element is added/retrieved, same procedure follows:  
  
  
a. Using key.hashCode() [ see above step 4],determine initial hashvalue for the key  
  
b. Pass intial hashvalue as hashValue  in    hash(hashValue) function, to calculate the final hashvalue.  
  
c. Final hash value is then passed as a first parameter in the indexFor(int ,int )method .  
    The second parameter is length which is a constant in HashMap Java Api , represented by                             DEFAULT\_INITIAL\_CAPACITY  
  
    The default  value of DEFAULT\_INITIAL\_CAPACITY is 16 in HashMap Java Api .  
  
 indexFor(int,int) method  returns the first entry in the appropriate bucket. The linked list in the bucket is then iterated over - (the end is found and the element is added or the key is matched and the value is returned )  
  
  
Explanation about indexFor(int,int) is below :

/\*\*

\* Returns index for hash code h.

\*/

static int indexFor(int h, int length) {

return h & (length-1);

}

The above function indexFor() works because Java HashMaps always have a capacity, i.e. number of buckets, as a power of 2.  
 Let's work with a capacity of 256,which is 0x100, but it could work with any power of 2. Subtracting 1  
from a power of 2 yields the exact bit mask needed to bitwise-and with the hash to get the proper bucket index, of range 0 to length - 1.  
256 - 1 = 255  
0x100 - 0x1 = 0xFF  
E.g. a hash of 257 (0x101) gets bitwise-anded with 0xFF to yield a bucket number of 1.  
  
  
  
Interviewer:    What if  when two  keys are same and have the same hashcode ?  
If key needs to be inserted and already inserted hashkey's hashcodes are same, and keys are also same(via reference or using equals() method)  then override the previous key value pair with the current key value pair.  
  
The other important point to note is that in Map ,Any class(String etc.) can serve as a key if and only if it overrides the equals() and hashCode() method .

Interviewer:  How will you measure the performance of HashMap?  
  
According to [Oracle Java docs](http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html),    
  
An instance of HashMap has two parameters that affect its performance: initial capacity and load factor.   
  
The capacity is the number of buckets in the hash table( HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.), and the initial capacity is simply the capacity at the time the hash table is created.   
  
  
The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.  
  
In HashMap class, the default value of load factor is (.75) .  
  
Interviewer : What is the time complexity of Hashmap get() and put() method ?  
  
The hashmap implementation provides constant time performance for (get and put) basic operations  
i.e the complexity of get() and put() is O(1) , assuming the hash function disperses the elements properly among the buckets.   
  
Interviewer : What is the difference between HashMap and ConcurrentHashMap ?  
  
It is also one of the popular interview question on HashMap , you can find the answer here  
[HashMap vs ConcurrentHashMap](https://javahungry.blogspot.in/2014/02/hashmap-vs-concurrenthashmap-java-collections-interview-question.html)

How HashMap works in Java 8

In java 8 there is a lot of changes in the inner representation of HashMap. The implementation of HashMap went from 1k lines of code in java 7 to 2k lines of code in java 8. In java 8, Node class contains the exact same information as the Entry class i.e Node class contains ( hash , key, value, bucketindex).  
Here is the implementation of the Node class in java 8.

static class Node<K,V> implements Map.Entry<K,V> {

final int hash;

final K key;

V value;

Node<K,V> next;

Node(int hash, K key, V value, Node<K,V> next) {

this.hash = hash;

this.key = key;

this.value = value;

this.next = next;

}

What’s the big difference from java 7 ?  
  
TreeNode class extends Node through LInkedHashMap.Entry<K,V>. In other words ,  
TreeNode indirectly extends Node class. A TreeNode internally implements Red-Black tree structure. It stores more information than Node class so that it can perform get(),add() or delete() operations in O(log(n)). As we know Node class contains (hash,key,value,bucketindex) where as TreeNode class contains following list of data.

static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {

TreeNode<K,V> parent; // red-black tree links

TreeNode<K,V> left;

TreeNode<K,V> right;

TreeNode<K,V> prev; // needed to unlink next upon deletion

boolean red;

TreeNode(int hash, K key, V val, Node<K,V> next) {

super(hash, key, val, next);

}

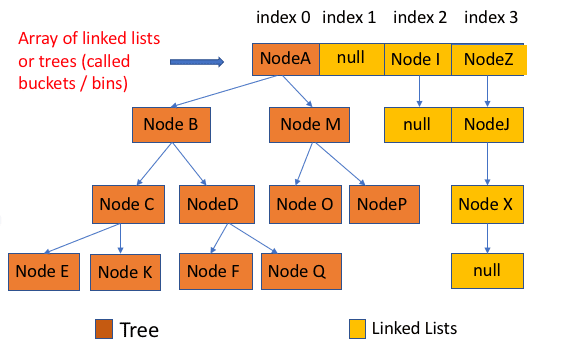
What are Red Black Trees and Why they are used?  
  
According to [Wikipedia](https://en.wikipedia.org/wiki/Red%E2%80%93black_tree),  
Red-Black trees are self-balancing binary search trees. Red-black tree makes sure that the length of the binary search trees is always log(n) despite new addition or removal of nodes. The main advantage of using Red-black tree structure is in a case where many entries are in the same bucket. For search operation,in java 7,it will take O(n) with a linked list. While in java 8 , the same search operation in a tree will cost O(log(n)).  
  
Drawbacks : Tree really takes more space than the linked list.  
  
By Inheritance, bucket can contain both Node(Entry object) and TreeNode(Red-black tree).  
  
Oracle java developers decided to use both data structures and following rules are applied.  
  
1. If for a given bucket , there are more than 8 Nodes, the linked list is converted into a  
red-black tree. This is represented by the following code in HashMap class :

static final int TREEIFY\_THRESHOLD = 8;

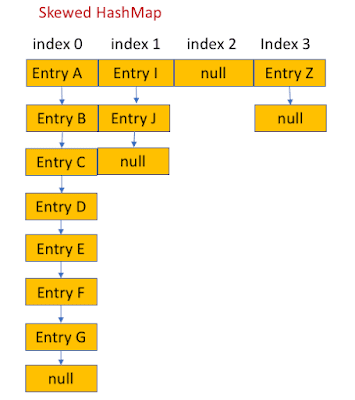
2. If for a given bucket , there are less than 6 nodes, then the tree can be converted  
into a linkedlist.

static final int UNTREEIFY\_THRESHOLD = 6;

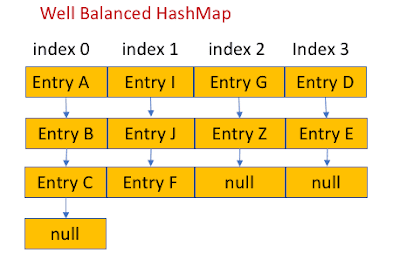
The below Java 8 HashMap image shows both trees(at bucket 0) and linkedlists (at bucket 1,2 and 3). Bucket 0 is a Tree because it contains at least 8 nodes.

[](https://3.bp.blogspot.com/-btSv072MD00/XEX6bB3VNMI/AAAAAAAAA70/wuPl7sjIGbIuuhkDAro9PSsnm28-biPbQCLcBGAs/s1600/Tree+and+LinkedList+together+in+a+HashMap+bucket.png)

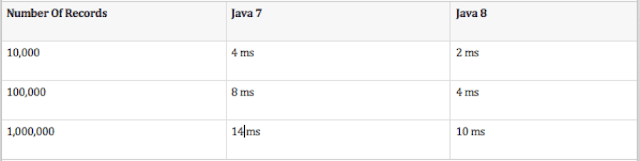
Performance Issue in Java 8  
  
In the best case scenario, put() and get() operations have a O(1) time complexity. But if you do not provide efficient hash function of the key , then you might end up with very slow get() and put() operations.  
  
The good performance of the get() and put() operations depend on the repartition of the data into the different indexes of the bucket.  
If the hash function of your key is poorly designed, then you will have a skew repartition (capacity of the bucket becomes irrelevant). All the get() and put() operations that use the biggest linked lists of entry will be really slow. It is due to the reason as the get() and put() operation need to iterate the entire lists. In the worst case scenario (when hash function of the key is poorly designed and all the entry objects are in the same buckets), you would end up with O(n) time complexity.  
  
Below are the images showing skewed HashMap and well balanced HashMap. In the case of skewed HashMap . the put() and get() operations on the bucket 0 are costlier. Getting the Entry F will cost 5 iterations.

[](https://3.bp.blogspot.com/-s9KOj3LxtOI/XEX8AfTTNsI/AAAAAAAAA8A/WrWKfb22Ar89vII1fbP9jjm87zuIXXB2QCLcBGAs/s1600/skewed+HashMap.png)

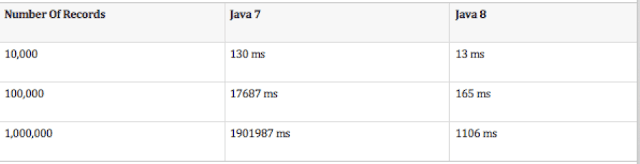
In the case of well balanced HashMap , getting the Entry F will cost 2 iterations.

[](https://3.bp.blogspot.com/-0qjMCodyPnY/XEX8Rxm85EI/AAAAAAAAA8I/4niC7OSgYJcPnLxHU89pEk7Uxj78b7YDgCLcBGAs/s1600/Well+balanced+HashMap.png)

Interestingly, both above HashMaps store the same amount of data and have the same bucket size.  
  
Why there is difference between skewedHashMap and well balanced HashMap entries  
The only difference is the hash function of the key that distributes the Entries in the buckets.  
  
Testing : Performance of HashMap Java 7 vs Java 8 :  
  
 1. get() method with proper hashCode() logic

[](https://2.bp.blogspot.com/-sL7Ged_Ncw0/XEYCKPtPxfI/AAAAAAAAA8s/GNoH6UYYQogsrTwWhQrA5RRl7cQm-3HjwCLcBGAs/s1600/HashMap+get+performance+java7+vs+java8.png)

 1. get() method with poorly designed hashCode() logic

[](https://2.bp.blogspot.com/-HacBaiunnAg/XEYCjwmeEhI/AAAAAAAAA80/q_fzv9xyvR8ewk-tqxNDK-16TTNXC65mQCLcBGAs/s1600/HashMap+get+performance.png)

When using HashMap, your goal is to write a hash function for your keys that spreads the keys into most number of possible buckets. To do that, you need to avoid hash collisions.  
  
Memory Overhead in Java 8 and Java 7  
  
With the java 8 implementation of HashMap, it is quite difficult to get the memory usage because a Node object can contain the same data as an Entry object or same data plus a boolean and 4 more references (if it’s TreeNode).  
If all the nodes are Nodes then memory overhead in Java 8 will be same as the Java 7 HashMap.  
The worst case scenario , if all the nodes in the HashMap are TreeNodes , then the memory overhead of a Java 8 HashMap becomes :

N \* sizeOf(integer) + N \* sizeOf(boolean) + sizeOf(reference)\* (9\*N+CAPACITY)

Resizing overhead in Java 8 and Java 7  
  
If you need to store large amount of data into the HashMap then you should create a  
HashMap with an initial capacity close to your expected volume.  
  
If you missed that , the HashMap will take the default size of 16 and load factor of 0.75. The first 11 put() operations will be fast but the 12th (16\*0.75) will resize the bucket with a new capacity of 32. The 13th to 23rd will be fast but 24th (0.75 \* 32) will again recreate a costly new representation that doubles the initial size of the bucket. The initial resizing operation will appear at 48th , 96th ,192nd , 384th call of put() operation.  
At low volume of data the full recreation of the bucket is fast but at high volume of data it can take seconds to minutes.  
  
Note : By setting the expected size initially, you can avoid these costly operations.  
  
Drawback : If you initialize HashMap with size 2^32 but you are using only 2^29 buckets then you will waste a lot of memory.  
  
This is the most popular question in the java developer role. You should at least familiar with How HashMap works in java. For performance wise you will not see the difference between a O(1) and O(n) or O(log(n)). During performance testing, it becomes important to know how HashMap works and to understand the importance of the hash function of the key. Please mention in comments if you have any questions.  
  
Source of image : http://ru.wikipedia.org

**About The Author**

**keyset**() **vs entrySet**() in map

# HashMap vs LinkedHashMap

**HashMap and LinkedHashMap** are two of the most common used Map implementation in Java. Main difference between HashMap **and LinkedHashMap** is that **LinkedHashMap** maintains insertion order of keys, order in which keys are inserted in to **LinkedHashMap**. On the other hand **HashMap** doesn't maintain any order or keys or values.

# Interview Programming

## Find The First Non Repeated Character In A String : Technical Interview Question

https://ide.geeksforgeeks.org/hpLtmRRj9R

// Java program to read data of various types using Scanner class.

import java.util.\*;

import java.io.\*;

public class A

{

public static void main(String[] args)

{

String a= "anand";

HashMap<Character,Integer> hm=new HashMap<Character,Integer>();

int l=a.length();

for(int i=0; i<l; i++)

{

Character s= a.charAt(i);

if(hm.containsKey(s))

{

hm.put(s, hm.get(s)+1);

}

else{

hm.put(s,1);

}

}

//Set<Map.Entry<Character, Integer>> k = hm.entrySet();

for(Map.Entry<Character,Integer> m:hm.entrySet())

{

if(m.getValue()==1)

System.out.println(m.getKey());

}

}

}

## **Q #1) Write a Java Program to reverse a string without using String inbuilt function.**

import java.io.\*;

class GFG {

public static void main (String[] args) {

String s1="mynameissantosh";

StringBuilder s2=new StringBuilder();

s2.append(s1);

s2.reverse();

System.out.println(s2);

}

}

## **Q #2) Write a Java Program to reverse a string without using String inbuilt function reverse().**

/\*package whatever //do not write package name here \*/

Method-1

import java.io.\*;

class GFG {

public static void main (String[] args) {

String s1="mynameissantosh";

String[] s2=s1.split("");

// String[] s3=new String[s2.length];

StringBuffer s3=new StringBuffer();

for(int i=s2.length-1;i>=0;i--)

{

s3=s3.append(s2[i]);

}

System.out.println(s3);

}

}

Method-2

/\*package whatever //do not write package name here \*/

import java.io.\*;

class GFG {

public static void main (String[] args) {

String s1="mynameissantosh";

char[] c1=s1.toCharArray();

for(int i=c1.length-1;i>=0;i--)

{

System.out.print(c1[i]);

}

}

}

Method-3

/\*package whatever //do not write package name here \*/

import java.io.\*;

class GFG {

public static void main (String[] args) {

String s1="mynameissantosh";

String[] s2=s1.split("");

for(int i=s2.length-1;i>=0;i--)

{

System.out.print(s2[i]);

}

}

}

## **Q #3) Write a Java Program to swap two numbers with using the third variable.**

/\*package whatever //do not write package name here \*/

import java.io.\*;

class GFG {

public static void main (String[] args) {

int a=2,b=3,c=0;

System.out.print(a+" "+b);

c=a;

a=b;

b=c;

System.out.print(a+" "+b);

}

}

## **Q #4) Write a Java Program to swap two numbers without using the third variable.**

/\*package whatever //do not write package name here \*/

import java.io.\*;

class GFG {

public static void main (String[] args) {

int a=2,b=3;

System.out.print(a+" "+b+"\n");

a=a-b;

b=a+b;

a=b-a;

System.out.print(a+" "+b);

}

}

## **Q #5) Write a Java Program to count the number of words in a string using HashMap.**

**/\*package whatever //do not write package name here \*/**

**import java.io.\*;**

**import java.util.\*;**

**class GFG {**

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

**Scanner sc=new Scanner(System.in);**

**String s1=sc.nextLine();**

**Map<String,Integer> hm=new HashMap<String,Integer>();**

**String[] s2=s1.split(" ");**

**for(int i=0;i<=s2.length-1;i++)**

**{**

**if(hm.containsKey(s2[i]))**

**{**

**hm.put(s2[i],hm.get(s2[i])+1);**

**}**

**else**

**hm.put(s2[i],1);**

**}**

**hm.forEach((k,v)->System.out.println(k+" "+v));**

**// System.out.print(a+" "+b);**

**}**

**}**

## **Write a Java Program to iterate HashMap using While and advance for loop.**

/\*package whatever //do not write package name here \*/

import java.io.\*;

import java.util.\*;

class GFG {

public static void main (String[] args) {

Scanner sc=new Scanner(System.in);

String s1=sc.nextLine();

Map<String,Integer> hm=new HashMap<String,Integer>();

String[] s2=s1.split(" ");

for(int i=0;i<=s2.length-1;i++)

{

if(hm.containsKey(s2[i]))

{

hm.put(s2[i],hm.get(s2[i])+1);

}

else

hm.put(s2[i],1);

}

//itterating using while loop

Iterator itr=hm.entrySet().iterator();

while(itr.hasNext())

{

Map.Entry m=(Map.Entry)itr.next();

System.out.println(m.getKey()+""+m.getValue());

}

// using for loop

for(Map.Entry<String,Integer>h:hm.entrySet())

{

System.out.println(h.getKey()+""+h.getValue());

}

// directly

System.out.print(hm);

//using for each lambda java8

hm.forEach((k,v)->System.out.println(k+" "+v));

// System.out.print(a+" "+b);

}

}

## **Q #7) Write a Java Program to find whether a number is prime or not.**

/\*package whatever //do not write package name here \*/

import java.io.\*;

import java.util.\*;

class GFG {

public static void main (String[] args) {

Scanner sc=new Scanner(System.in);

int a=sc.nextInt();

int b=a/2,p=0;

if((a==0) || (a==1))

{

System.out.print("no is not prime");

}

else

{

for(int i=2;i<=b;i++)

{

if(a%i==0)

p=1;

}

if(p==1)

System.out.print("no is not prime");

else

System.out.print("no is prime");

}

}

}

## **Q #8) Write a Java Program to find whether a string or number is palindrome or not.**

**/\*package whatever //do not write package name here \*/**

**import java.io.\*;**

**import java.util.\*;**

**class GFG {**

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

**Scanner sc=new Scanner(System.in);**

**String t="";**

**String s1=sc.nextLine();**

**String[] s2=s1.split("");**

**for(int i=s2.length-1;i>=0;i--)**

**{**

**t=t+ s2[i];**

**}**

**if(s1.equals(t))**

**{**

**System.out.println("String is pallindrome");**

**}**

**else**

**{**

**System.out.println("String is not pallindrome");**

**}**

## 

**}**

**}**

**Q #9) Write a Java Program for Fibonacci series.**

/\*package whatever //do not write package name here \*/

import java.io.\*;

import java.util.\*;

class GFG {

public static void main (String[] args) {

Scanner sc=new Scanner(System.in);

int a=sc.nextInt();

int[] b=new int[a];

b[0]=0;

b[1]=1;

for(int i=2;i<a;i++)

{

b[i]=b[i-1]+b[i-2];

}

for(int j=0;j<a;j++)

{

System.out.print(b[j]+" ");

}

}

}

## **Q #10) Write a Java Program to iterate ArrayList using for-loop, while-loop, and advance for-loop.**

**/\*package whatever //do not write package name here \*/**

**import java.io.\*;**

**import java.util.\*;**

**class GFG {**

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

**List<Integer> l1=new ArrayList<Integer>();**

**l1.add(12);**

**l1.add(13);**

**l1.add(14);**

**l1.add(15);**

**Iterator itr=l1.iterator();**

**while(itr.hasNext())**

**{**

**System.out.println(itr.next()+" ");**

**}**

**for(Integer i:l1)**

**{**

**System.out.println(i);**

**}**

**for(int i=0;i<l1.size();i++)**

**{**

**System.out.println(l1.get(i));**

**}**

**}**

**}**

Or

/\*package whatever //do not write package name here \*/

import java.io.\*;

import java.util.\*;

class GFG {

public static void main (String[] args) {

ArrayList l1=new ArrayList();// it will give run time warning

l1.add(12);

l1.add(13);

l1.add(14);

l1.add(15);

Iterator itr=l1.iterator();

while(itr.hasNext())

{

System.out.println(itr.next()+" ");

}

for(Object i:l1)

{

System.out.println(i);

}

for(int i=0;i<l1.size();i++)

{

System.out.println(l1.get(i));

}

}

}

## **Q #16) Write a Java Program to find the second highest number in an array.**

**Answer:** In this program, we have initialized an array with 10 random elements out of which we are going to find the second highest number. Here, we have two integers- the largest and second largest. Both set to the first index of the element. Then, we have printed all the elements using for loop.

Now the logic is when the element at the 0th index is greater than the largest, then assign arr[0] to largest and secondLargest to largest. Again, if the element at the 0th index is greater than the secondLargest, then assign secondLargest to arr[0].

This will be repeated for each iteration and ultimately after comparing or completing iterations up to array length will give you the secondLargest element.

|  |  |
| --- | --- |
| 1 | package codes; |
| 2 | public class SecondHighestNumberInArray { | |

|  |  |  |
| --- | --- | --- |
| 3 | public static void main(String[] args) | |
| 4 | { |

|  |  |  |
| --- | --- | --- |
| 5 | int arr[] = { 100,14, 46, 47, 94, 94, 52, 86, 36, 94, 89 }; | |
| 6 | int largest = 0; |

|  |  |
| --- | --- |
| 7 | int secondLargest = 0; |
| 8 | System.out.println("The given array is:"); | |

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | for (int i = 0; i < arr.length; i++) | | |
| 10 | | { |

|  |  |  |
| --- | --- | --- |
| 11 | System.out.print(arr[i] + "\t"); | |
| 12 | } |

|  |  |  |
| --- | --- | --- |
| 13 | for (int i = 0; i < arr.length; i++) | |
| 14 | { |

|  |  |  |
| --- | --- | --- |
| 15 | if (arr[i] > largest) | |
| 16 | { |

|  |  |  |
| --- | --- | --- |
| 17 | secondLargest = largest; | |
| 18 | largest = arr[i]; |

|  |  |
| --- | --- |
| 19 | } |
| 20 | else if (arr[i] > secondLargest) | |

|  |  |
| --- | --- |
| 21 | { |
| 22 | secondLargest = arr[i]; | |

|  |  |  |
| --- | --- | --- |
| 23 | } | |
| 24 | } |

|  |  |  |
| --- | --- | --- |
| 25 | System.out.println("\nSecond largest number is:" + secondLargest); | |
| 26 | System.out.println("Largest Number is: "  +largest); |

|  |  |  |
| --- | --- | --- |
| 27 | } | |
| 28 | } |

**Output:**

The given array is:  
100 14 46 47 94 94 52 86 36 94 89  
Second largest number is:94  
Largest Number is: 100

## Q #17) Write a Java Program to check Armstrong number.

**Answer:** First of all we need to understand what Armstrong Number is. Armstrong number is the number which is the sum of the cubes of all its unit, tens and hundred digits for three-digit number.

153 = 1\*1\*1 + 5\*5\*5 + 3\*3\*3 = 1 + 125 + 27 = 153

If you have a four-digit number lets say

1634 = 1\*1\*1\*1 + 6\*6\*6\*6 + 3\*3\*3\*3 + 4\*4\*4\*4 = 1 + 1296 + 81 + 256 = 1634

Now, in this program, we have a temp and integers declared. We have initialized c with value 0. Then, we need to assign the integer value which we are going to check for Armstrong (in our case, let us say 153). Then we have assigned our temp variable with that number which we are going to check.

Thereafter, we have used while conditional check where the remainder is assigned to a and the number is divided by 10 and assigned to n. Now, our c variable which was set to zero initially is assigned with c+(a\*a\*a). Suppose we have to evaluate a four-digit number then c should be assigned with c + (a\*a\*a\*a).

Lastly, we have put an if-else statement for conditional checking where we have compared the value contained in c against temp(which has the actual number stored at this point). If it matches, then the number is Armstrong otherwise not.

|  |  |
| --- | --- |
| 1 | class Armstrong{ |
| 2 | public static void main(String[] args)  { | |

|  |  |
| --- | --- |
| 3 | int c=0,a,temp; |
| 4 | int n=153;//It is the number to check Armstrong | |

|  |  |
| --- | --- |
| 5 | temp=n; |
| 6 | while(n>0) | |

|  |  |
| --- | --- |
| 7 | { |
| 8 | a=n%10; | |

|  |  |  |
| --- | --- | --- |
| 9 | n=n/10; | |
| 10 | | c=c+(a\*a\*a); | |

|  |  |
| --- | --- |
| 11 | } |
| 12 | if(temp==c) | |

|  |  |  |
| --- | --- | --- |
| 13 | System.out.println("armstrong number"); | |
| 14 | else |

|  |  |
| --- | --- |
| 15 | System.out.println("Not armstrong number"); |

## Q #18) Write a Java Program to remove all white spaces from a string with using replace().

**Answer:** This is a simple program where we have our string variable str1.

Another string variable str2 is initialized with the replaceAll option which is an inbuilt method to remove n number of whitespaces. Ultimately, we have printed str2 which has no whitespaces.

|  |  |  |
| --- | --- | --- |
| 1 | class RemoveWhiteSpaces | |
| 2 | { |

|  |  |  |
| --- | --- | --- |
| 3 | public static void main(String[] args) | |
| 4 | { |

|  |  |  |
| --- | --- | --- |
| 5 | String str1 = "Saket Saurav        is a QualityAna    list"; | |
| 6 |  |

|  |  |  |
| --- | --- | --- |
| 7 | //1. Using replaceAll() Method | |
| 8 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | String str2 = str1.replaceAll("\\s", ""); | | |
| 10 | |  |

|  |  |  |
| --- | --- | --- |
| 11 | System.out.println(str2); | |
| 12 |  |

|  |  |  |
| --- | --- | --- |
| 13 | } | |
| 14 | } |

|  |  |
| --- | --- |
| 15 | } |

**Output:**

SaketSauravisaQualityAnalist

## Q #19) Write a Java Program to remove all white spaces from a string without using replace().

**Answer:**This is another approach to removing all white spaces. Again, we have one string variable str1 with some value. Then, we have converted that string into a character array using toCharArray().

Then, we have one StringBuffer object sb which will be used to append the value stored at chars[i] index after we have included for loop and one if condition.

If the condition is set such that then the element at i index of the character array should not be equal to space or tab. Finally, we have printed our StringBuffer object sb.

|  |  |  |
| --- | --- | --- |
| 1 | class RemoveWhiteSpaces | |
| 2 | { |

|  |  |  |
| --- | --- | --- |
| 3 | public static void main(String[] args) | |
| 4 | { |

|  |  |  |
| --- | --- | --- |
| 5 | String str1 = "Saket Saurav        is an Autom ation Engi ne      er"; | |
| 6 |  |

|  |  |  |
| --- | --- | --- |
| 7 | char[] chars = str1.toCharArray(); | |
| 8 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | StringBuffer sb = new StringBuffer(); | | |
| 10 | |  |

|  |  |  |
| --- | --- | --- |
| 11 | for (int i = 0; i < chars.length; i++) | |
| 12 | { |

|  |  |  |
| --- | --- | --- |
| 13 | if( (chars[i] != ' ') && (chars[i] != '\t') ) | |
| 14 | { |

|  |  |  |
| --- | --- | --- |
| 15 | sb.append(chars[i]); | |
| 16 | } |

|  |  |
| --- | --- |
| 17 | } |
| 18 | System.out.println(sb);           //Output : CoreJavajspservletsjdbcstrutshibernatespring | |

|  |  |  |
| --- | --- | --- |
| 19 | } | |
| 20 | } |

**Output:**

SaketSauravisanAutomationEngineer

# Interview Questions:

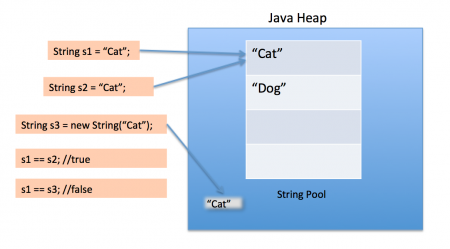
## Top 28 Most Frequently Asked Interview Core Java Interview Questions And Answers 2018

We are sharing 28 java interview questions , these questions are frequently asked by the recruiters.Java interview questions can be asked from any core java topic . So I try my best to provide you the java interview questions and answers for experienced which should be in your to do list before facing java questions in  technical interview  .  
 **Top 28  Java Interview Questions :**  
 **1. Which two method you need to implement for key Object in HashMap ?**

In order to use any object as Key in HashMap, it must implements equals and hashcode method in Java. Read [**How HashMap works in Java**](http://javahungry.blogspot.com/2013/08/hashing-how-hash-map-works-in-java-or.html)  for detailed explanation on how equals and hashcode method is used to put and get object from HashMap.   
  
**2. What is immutable object? Can you write immutable object?**Immutable classes are Java classes whose objects can not be modified once created. Any modification in Immutable object result in new object. For example is String is immutable in Java. Mostly Immutable are also final in Java, in order to prevent sub class from overriding methods in Java which can compromise Immutability. You can achieve same functionality by making member as non final but private and not modifying them except in constructor.

**3. What is the difference between creating String as new() and literal?**

When we create string with new() Operator, it’s created in heap and not added into string pool while String created using literal are created in String pool itself which exists in PermGen area of heap.



String s = new String("Test");  
   
does not  put the object in String pool , we need to call String.intern() method which is used to put  them into String pool explicitly. its only when you create String object as String literal e.g. String s = "Test" Java automatically put that into String pool.

**4. What is**[**difference between StringBuffer and StringBuilder**](http://javahungry.blogspot.com/2013/06/difference-between-string-stringbuilder.html)**in Java ?**

Classic Java questions which some people thing tricky and some consider very easy. StringBuilder in Java is introduced in Java 5 and only difference between both of them is that Stringbuffer methods are synchronized while StringBuilder is non synchronized. See [StringBuilder vs StringBuffer](http://javahungry.blogspot.com/2013/06/difference-between-string-stringbuilder.html) for more differences.

**5.  Write code to find the First non repeated character in the String  ?**

Another good Java interview question, This question is mainly asked by Amazon and equivalent companies. See [first non repeated character in the string : Amazon interview question](http://javahungry.blogspot.com/2013/12/first-non-repeated-character-in-string-java-program-code-example.html)

**6. What is the difference between ArrayList and Vector ?**  
This question is mostly used as a start up question in Technical interviews  on the topic of Collection framework . Answer is explained in detail here [Difference between ArrayList and Vector](http://javahungry.blogspot.com/2013/12/difference-between-arraylist-and-vector-in-java-collection-interview-question.html) .

**7. How do you handle error condition  while writing stored procedure or accessing stored procedure from java?**

This is one of the tough Java interview question and its open for all, my friend didn't know the answer so he didn't mind telling me. my take is that stored procedure should return error code if some operation fails but if stored procedure itself fail than catching SQLException is only choice.

**8. What is difference between Executor.submit() and Executer.execute() method ?**

*There is a difference when looking at exception handling. If your tasks throws an exception and if it was submitted with execute this exception will go to the uncaught exception handler (when you don't have provided one explicitly, the default one will just print the stack trace to System.err). If you submitted the task with submit any thrown exception, checked exception or not, is then part of the task's return status. For a task that was submitted with submit and that terminates with an exception, the Future.get will re-throw this exception, wrapped in an ExecutionException.*

**9. What is the difference between factory and abstract factory pattern?**

*Abstract Factory provides one more level of abstraction. Consider different factories each extended from an Abstract Factory and responsible for creation of different hierarchies of objects based on the type of factory. E.g. AbstractFactory extended by AutomobileFactory, UserFactory, RoleFactory etc. Each individual factory would be responsible for creation of objects in that genre.*

You can also refer What is Factory method design pattern in Java to know more details.

**10. What is Singleton? is it better to make whole method synchronized or only critical section synchronized ?**

[Singleton in Java is a class with just one instance in whole Java application](http://javahungry.blogspot.com/2013/08/singleton-design-pattern-use-in-java.html), for example java.lang.Runtime is a Singleton class. Creating Singleton was tricky prior Java 4 but once Java 5 introduced Enum its very easy. see my article [How to create thread-safe Singleton in Java](http://javahungry.blogspot.com/2013/08/singleton-design-pattern-use-in-java.html) for more details on writing Singleton using enum and double checked locking which is purpose of this Java interview question.

**11. Can you write critical section code for singleton?**

This core Java question is followup of previous question and expecting candidate to write Java singleton using double checked locking. Remember to use volatile variable to make Singleton thread-safe.

**12. Can you write code for**[**iterating**](http://javahungry.blogspot.com/2013/06/difference-between-iterator-and-enumeration-collections-java-interview-question-with-example.html)**over HashMap in Java 7 and Java 8 ?**

Tricky one but he managed to write using while and for loop. You can find the answer here [How to iterate or loop over HashMap in Java with Example](http://javahungry.blogspot.com/2017/11/how-to-iterate-or-loop-over-hashmap-in-java-with-example.html).  
  
**13. When do you override hashcode and equals() ?**  
Whenever necessary especially if you want to do equality check or want to use your object as key in HashMap.  
  
**14. What will be the problem if you don't override hashcode() method ?**  
You will not be able to recover your object from hash Map if that is used as key in HashMap.  
See here  [How HashMap works in Java](http://javahungry.blogspot.com/2013/08/hashing-how-hash-map-works-in-java-or.html) for detailed explanation.  
  
**15. Is it better to synchronize critical section of getInstance() method or whole getInstance() method ?**  
Answer is critical section because if we lock whole method than every time some one call this method will have to wait even though we are not creating any object)  
  
**16. What is the difference when String is gets created using literal or new() operator ?**  
When we create string with new() its created in heap and not added into string pool while String created using literal are created in String pool itself which exists in Perm area of heap.  
  
**17. Does not overriding hashcode() method has any performance implication ?**  
This is a good question and open to all , as per my knowledge a poor hashcode function will result in frequent collision in HashMap which eventually increase time for adding an object into Hash Map.  
  
**18. What’s wrong using HashMap in multithreaded environment? When get() method go to infinite loop ?**  
Another good question. His answer was during concurrent access and re-sizing.  
  
**19.  What do you understand by thread-safety ? Why is it required ? And finally, how to achieve thread-safety in Java Applications ?**  
  
Java Memory Model defines the legal interaction of threads with the memory in a real computer system. In a way, it describes what behaviors are legal in multi-threaded code. It determines when a Thread can reliably see writes to variables made by other threads. It defines semantics for volatile, final & synchronized, that makes guarantee of visibility of memory operations across the Threads.  
  
Let's first discuss about Memory Barrier which are the base for our further discussions. There are two type of memory barrier instructions in JMM - read barriers and write barrier.  
  
A read barrier invalidates the local memory (cache, registers, etc) and then reads the contents from the main memory, so that changes made by other threads becomes visible to the current Thread.  
A write barrier flushes out the contents of the processor's local memory to the main memory, so that changes made by the current Thread becomes visible to the other threads.  
**JMM semantics for synchronized**  
When a thread acquires monitor of an object, by entering into a synchronized block of code, it performs a read barrier (invalidates the local memory and reads from the heap instead). Similarly exiting from a synchronized block as part of releasing the associated monitor, it performs a write barrier (flushes changes to the main memory)  
Thus modifications to a shared state using synchronized block by one Thread, is guaranteed to be visible to subsequent synchronized reads by other threads. This guarantee is provided by JMM in presence of synchronized code block.  
  
**JMM semantics for Volatile  fields**  
Read & write to volatile variables have same memory semantics as that of acquiring and releasing a monitor using synchronized code block. So the visibility of volatile field is guaranteed by the JMM. Moreover afterwards Java 1.5, volatile reads and writes are not reorderable with any other memory operations (volatile and non-volatile both). Thus when Thread A writes to a volatile variable V, and afterwards Thread B reads from variable V, any variable values that were visible to A at the time V was written are guaranteed now to be visible to B.

Let's try to understand the same using the following code

Data data = null;

volatile boolean flag = false;

Thread A

-------------

data = new Data();

flag = true; <-- writing to volatile will flush data as well as flag to main memory

Thread B

-------------

if(flag==true){ <-- as="" barrier="" data.="" flag="" font="" for="" from="" perform="" read="" reading="" volatile="" well="" will="">

use data; <!--- data is guaranteed to visible even though it is not declared volatile because of the JMM semantics of volatile flag.

}

**20.  What will happen if you call return statement or System.exit on try or catch block ? will finally block execute?**

This is a very *popular tricky Java question* and its tricky because many programmer think that finally block always executed. This question challenge that concept by putting return statement in try or catch block or calling System.exit from try or catch block. Answer of this tricky question in Java is that finally block will execute even if you put return statement in try block or catch block but finally block won't run if you call System.exit form try or catch.

**21. Can you override private or static method in Java ?**

Another popular Java tricky question, As I said method overriding is a good topic to ask trick questions in Java.  Anyway, you can not override private or static method in Java, if you create similar method with same return type and same method arguments that's called method hiding.

**22. What will happen if we put a key object in a HashMap which is already there ?**

This tricky Java questions is part of [How HashMap works in Java](http://javahungry.blogspot.com/2013/08/hashing-how-hash-map-works-in-java-or.html), which is also a popular topic to create confusing and tricky question in Java. well if you put the same key again than it will replace the old mapping because HashMap doesn't allow duplicate keys.

**23. If a method throws NullPointerException in super class, can we override it with a method which throws RuntimeException?**

One more tricky Java questions from overloading and overriding concept. Answer is you can very well throw super class of RuntimeException in overridden method but you can not do same if its checked Exception.

**24. What is the issue with following implementation of compareTo() method in Java**

public int compareTo(Object o){

   Employee emp = (Employee) emp;

   return this.id - o.id;

}

**25. How do you ensure that N thread can access N resources without deadlock**

If you are not well versed in writing multi-threading code then this is real tricky question for you. This Java question can be tricky even for experienced and senior programmer, who are not really exposed to deadlock and race conditions. Key point here is order, if you acquire resources in a particular order and release resources in reverse order you can prevent deadlock.

**26. What is difference between CyclicBarrier and CountDownLatch in Java**

Relatively newer Java tricky question, only been introduced form Java 5. Main difference between both of them is that you can reuse CyclicBarrier even if Barrier is broken but you can not reuse CountDownLatch in Java. See CyclicBarrier vs CountDownLatch in Java for more differences.

**27. Can you access non static variable in static context?**

Another tricky Java question from Java fundamentals. No you can not access static variable in non static context in Java. Read why you can not access non-static variable from static method to learn more about this tricky Java questions.  
  
**28. What is the difference between sleep() and wait() method?**  
sleep() does not release the lock while wait method release the lock. sleep() method is present in java.lang.Thread class while wait() method  is present in java.lang.Object class. For more differences  
please check [difference between sleep and wait method](http://javahungry.blogspot.com/2015/11/5-difference-between-sleep-and-wait-with-example.html).

## Top 40 Java Web Services Interview Questions And Answers

**Q1  What are web services ?**  
  
According to [oracle docs](http://docs.oracle.com/javaee/6/tutorial/doc/gijvh.html), web services can be defined as

*Web services are client and server applications that communicate over the World Wide Web’s (WWW) HyperText Transfer Protocol (HTTP). Web services provide a standard means of inter operating between software applications running on a variety of platforms and frameworks.*

Main characteristics of the Web Services  are :  
  
1. Interoperability  
2. Extensibility  
3. Machine processable descriptions.  
  
for example in simple words , when we call somebody so the person dialing and calling is the client  application , while person receiving the call is server application and "hello" word is the protocol as similar to HTTP request .  
  
  
**Q2** **What is the difference between SOA and a web service?**

SOA (Service-Oriented Architecture) is an architectural pattern that makes possible for

services to interact with one another independently.

Web Services is a realization of SOA concept, that leverages XML, JSON, etc. and common Internet protocols such as HTTP(S), SMTP, etc.

SOA is a system-level architectural style that tries to expose business. WOA is an interface-level architectural style that focuses on the means by which these service capabilities are exposed to consumers.  
 **Q3 What is SOAP?**  
*SOAP* *(*Simple Object Access Protocol*)*is a transport protocol for sending and receiving requests and responses on XML format, which can be used on top of transport protocols such as HTTP, SMTP, UDP, etc.

**Q4** **What is REST?**

REST (REpresentational State Transfer) is an architectural style by which data can be transmitted over transport protocol such as HTTP(S).  
 **Q5  What is the difference between a REST web service and a SOAP web service?**

Below are the main differences between REST and SOAP web service

* REST supports different formats like text, JSON and XML; SOAP only supports XML;
* REST works only over HTTP(S) on a transport layer; SOAP can be used different protocols on a transport layer;
* REST works with resources, each unique URL is some representation of a resource; SOAP works with operations, which implement some business logic through different interfaces;
* SOAP based reads can’t be cached, for SOAP need to provide caching; REST based reads can be cached;
* SOAP supports SSL security and WS-security(Web Service-security); REST only supports SSL security;
* SOAP supports ACID (Atomicity, Consistency, Isolation, Durability); REST supports transactions, but it is neither ACID compliant nor can provide two phase commit.

**Q6 How  to decide which one of web service to use REST or SOAP?**  
  
“REST vs SOAP” we can rephrased to "Simplicity vs Standard". Of course, "Simplicity" with REST at most cases wins, it wins in performance, scalability and support for multiple data formats, but SOAP is favored where service requires comprehensive support for security (WS-security) and transactional safety (ACID).

**“SOAP”**  
 **Q7  What is WSDL?**

WSDL (Web Services Description Language) is an XML format for describing web services and how to access them.

**Q8  What is JAX-WS?**

JAX-WS (Java API for XML Web Services) is a set of APIs for creating web services in XML format.

**Q9 What is JAXB?**

JAXB (Java Architecture for XML Binding) is a Java standard that defines how Java objects are converted from and to XML. It makes reading and writing of XML via Java relatively easy.

**Q10 Can we send soap messages with attachments?**

Yes, we can send different formats such as PDF document, image or other binary file with soap messages as an attachment. Messages send using the binary data. SOAP messages is attached with MIME extensions that come in multipart/related.

An example:

MIME-Version: 1.0

Content-Type: Multipart/Related; boundary=MIME\_boundary; type=text/xml;

        start="<claim061400a.xml@ javahungry.com>"

Content-Description: This is the optional message description.

<?xml version='1.0' ?>

<SOAP-ENV:Envelope

xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">

<SOAP-ENV:Body>

..

<theSignedForm href="cid:claim061400a.tiff@javahungry.com"/>

..

</SOAP-ENV:Body>

</SOAP-ENV:Envelope>

--MIME\_boundary

Content-Type: image/tiff

Content-Transfer-Encoding: binary

Content-ID: <claim061400a.tiff@javahungry.com>

...binary TIFF image...

--MIME\_boundary—  
 **Q11 What is MTOM?**

MTOM (Message Transmission Optimization Mechanism) is a mechanism for transmitting large binary attachments with SOAP messages as raw bytes, allowing for smaller messages.

**Q12 What is XOP?**

XOP (XML-binary Optimized Packaging) is a mechanism defined for the serialization of XML Information Sets that contain binary data, as well as deserialization back into the XML Information Set.

**Q13 What is a SOAP envelope element?**

SOAP envelop element is the root element of a SOAP message which defines the XML document as a SOAP message.

An example:

<?xml version="1.0"?>  
<soap:Envelope  
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"  
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">  
  ...  
  Message information  
  ...  
</soap:Envelope>  
 **Q14 What does a SOAP namespace defines?**

SOAP *namespace*defines the Envelope as a *SOAP* Envelope.

An example:

xmlns:soap=http://www.w3.org/2001/12/soap-envelope

**Q15 What is the SOAP encoding?**

SOAP encoding is a method for structuring the request which is suggested within the SOAP specification, known as the SOAP serialization.

**Q16 What does SOAP encodingStyle attribute defines?**

SOAP encodingStyle defines the serialization rules used in a SOAP message. This attribute may appear on any element, and is scoped to that element's contents and all child elements not themselves containing such an attribute. There is no default encoding defined for a SOAP message.

An example:

SOAP-ENV:encodingStyle="http://www.w3.org/2001/12/soap-encoding"

**Q17 What are 2 styles web service’s endpoint by using JAX-WS?**

* RPC (remote procedure call) style web service in JAX-WS;
* document style web service in JAX-WS.

**Q18 What is encoding rules for header entries?**

* a header entry is identified by its fully qualified element name, which consists of the namespace URI and the local name. All immediate child elements of the SOAP Header element must be namespace-qualified.
* the SOAP encodingStyle attribute may be used to indicate the encoding style used for the header entries.
* the SOAP mustUnderstand attribute and SOAP actor attribute may be used to indicate how to process the entry and by whom.

**Q19 What is the wsimport tool?**

The wsimport tool is used to parse an existing Web Services Description Language (WSDL) file and generate required files (JAX-WS portable artifacts) for web service client to access the published web services: https://docs.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html

**Q20 What is the wsgen tool?**

The wsgen tool is used to parse an existing web service implementation class and generates required files (JAX-WS portable artifacts) for web service deployment: http://docs.oracle.com/javase/6/docs/technotes/tools/share/wsgen.html

* What the tool are required to test SOAP services?

      SOAPUI tool for SOAP WS: http://www.soapui.org/

**Q21 What is the difference between SOAP and other remote access techniques?**

* SOAP is simple to use and it is non - symmetrical unlike DCOM or CORBA is highly popular and usually have complexity in them.
* SOAP provides greater platform independent with the language independence unlike DCOM or CORBA doesn't provide any of these.
* SOAP uses HTTP as its transport protocol and the data are being saved in XML format that can be ready by human, whereas DCOM or CORBA have their own binary formats that are used to transport the data in complicated manner.  
  SOAP identify the object other than URL endpoint. SOAP objects are stateless and it is hard to maintain that. Whereas, it is not hard to maintain in case of other remote access techniques.

**“REST”**

**Q22 What is a resource in a REST?**

      A resource is a unique URL with representation of an object which we can get contents via GET and modify via PUT, POST, DELETE.

**Q23 What are HTTP methods supported by REST?**

* GET;
* POST;
* PUT;
* DELETE;
* OPTIONS;
* HEAD.

**Q24 Whether can use GET request instead of POST to create a resource?**

It is not possibly, because **GET can’t change a resource.**

**Q25 What is the difference between PUT and POST?**

Need to use PUT when can update a resource completely through a specific resource. For example, if know that an article resides at http://javahungry.blogspot.com/article/123, can PUT a new resource representation of this article through a PUT on this URL. If do not know the actual resource location for instance, when add a new article, can use POST.

PUT is idempotent, while POST is not. It means if use PUT an object twice, it has no effect.

**Q26 What is WADL?**

WADL (Web Application Description Language) is a XML description of a deployed RESTful web application.

**Q27 What are frameworks available to implement REST web services?**

Jersey, Restlet, EasyRest, etc.

**Q28 What is the Restlet framework?**

Restlet is a lightweight, comprehensive, open source RESTful web API framework for the Java platform.

It has advantages such as

* websocket and server-sent events support;
* HTTP/2 support;
* transparent HTTP PATCH support;
* client cache service;
* fluent APIs.

http://restlet.com/

**Q29 What is the Jersey framework?**

Jersey is open source framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs and serves as a JAX-RS (JSR 311 & JSR 339) Reference Implementation. It has advantages such as

* contains support for Web Application Description Language (WADL);
* contains Jersey Test Framework which lets run and test Jersey REST services inside JUnit;
* supports for the REST MVC pattern, which would allow to return a View from Jersey services rather than just data.

https://jersey.java.net/

**Q30 What is the RESTeasy framework?**

RESTeasy is a JBoss project, which implements of the JAX-RS specification. It has benefits such as

* fully certified JAX-RS implementation; supports HTTP 1.1 caching semantics including cache revalidation;
* JAXB marshalling into XML, JSON, Jackson, Fastinfoset, and Atom as well as wrappers for maps, arrays, lists, and sets of JAXB Objects;
* OAuth2 and Distributed SSO with JBoss AS7;
* rich set of providers for: XML, JSON, YAML, Fastinfoset, Multipart, XOP, Atom, etc.

http://resteasy.jboss.org/

**Q31 What is the difference between AJAX and REST?**

* in Ajax, the request are sent to the server by using XMLHttpRequest objects; REST have a URL structure and a request/response pattern the revolve around the use of resources;
* Ajax eliminates the interaction between the customer and server asynchronously; REST requires the interaction between the customer and server;
* Ajax is a set of technology; REST is a type of software architecture and a method for users to request data or information from servers.

**Q32 What tool are required to test REST services?**

Firefox “poster” plugin for RESTFUL services. https://addons.mozilla.org/en-us/firefox/addon/poster/

[](https://1.bp.blogspot.com/-l_4Sb1tEk_c/VbSfpcirsbI/AAAAAAAAAo4/zqLjasVLwWw/s1600/java+web+services+interview+questions+and+answers.jpg)

**Q33 What does a @Path annotation do?**

     @Path annotation binds URI pattern to a Java method.

**import** **javax.ws.rs.GET**;

**import** **javax.ws.rs.Path**;

**import** **javax.ws.rs.core.Response**;

**@Path**("/persons")

public **class** **PersonRestService** {

**@GET**

public Response getPerson() {

**return** Response.status(**200**).entity("getPerson is called").build();

}

**@GET**

**@Path**("/vip")

public Response getPersonVIP() {

**return** Response.status(**200**).entity("getPersonVIP is called").build();

}

}

On calling URI: “/persons” result:  getPerson is called

On calling URI: “/persons/vip” result:  getPersonVIP is called

**Q34  What does a @PathParam do?**

     @PathParam annotation injects the value of URI parameter that defined in @Path expression.

**import** **javax.ws.rs.GET**;

**import** **javax.ws.rs.Path**;

**import** **javax.ws.rs.PathParam**;

**import** **javax.ws.rs.core.Response**;

**@Path**("/persons")

public **class** **PersonRestService** {

**@GET**

**@Path**("{id}")

public Response getPersonById(**@PathParam**("id") String id) {

**return** Response.status(**200**).entity("getPersonById is called, id : " + id).build();

}

}

On calling URI: “/persons/1” result:  getPersonById is called, id : 1

**Q35  What does a @QueryParam do?**

    @QueryParam annotation injects URI query parameter into Java method.

**import** **java.util.List**;

**import** **javax.ws.rs.GET**;

**import** **javax.ws.rs.Path**;

**import** **javax.ws.rs.QueryParam**;

**import** **javax.ws.rs.core.Response**;

**@Path**("/persons")

public **class** **PersonService** {

**@GET**

**@Path**("/query")

public Response getPersons(

**@QueryParam**("from") int from,

**@QueryParam**("to") int to,

**@QueryParam**("orderBy") List&lt;String&gt; orderBy) {

**return** Response

.status(**200**)

.entity("getPersons is called, from : " + **from** + ", to : " + to

+ ", orderBy" + orderBy.toString()).build();

}

}

On calling URI: “/persons/query?from=10&to=20&orderBy=age&orderBy=name” result: getPersons is called, from : 10, to : 20, orderBy**[**age, name**]**

**Q36  What does a @MatrixParam do?**

@MatrixParam are a set of **“name=value”** in URI path.

**import** **javax.ws.rs.GET**;

**import** **javax.ws.rs.MatrixParam**;

**import** **javax.ws.rs.Path**;

**import** **javax.ws.rs.PathParam**;

**import** **javax.ws.rs.core.Response**;

**@Path**("/books")

public **class** **BookService** {

**@GET**

**@Path**("{year}")

public Response getBooks(**@PathParam**("year") String year,

**@MatrixParam**("author") String author,

**@MatrixParam**("country") String country) {

**return** Response

.status(**200**)

.entity("getBooks is called, year : " + year

+ ", author : " + author + ", country : " + country)

.build();

}

}

On calling URI: “/books/2015” result: getBooks is called, year : 2015, author : null, country : null

On calling URI: “/books/2015;author= doyle;country=scotland” result: getBooks is called, year : 2015, author : doyle, country : scotland

**Q37  What does a @FormParam do?**

@FormParam bind HTML form parameters value to a Java method.

**import** **javax.ws.rs.FormParam**;

**import** **javax.ws.rs.POST**;

**import** **javax.ws.rs.Path**;

**import** **javax.ws.rs.core.Response**;

**@Path**("/persons")

public **class** **PersonService** {

**@POST**

**@Path**("/add")

public Response addPerson(

**@FormParam**("name") String name,

**@FormParam**("age") int age) {

**return** Response.status(**200**)

.entity("addPerson is called, name : " + name + ", age : " + age)

.build();

}

}

HTML form:

<html>

<body>

<form action="/persons/add" method="post">

<p>

Name : <input type="text" name="name" />

</p>

<p>

Age : <input type="text" name="age" />

</p>

<input type="submit" value="Add Person" />

</form>

</body>

</html>

**Q39  How to get HTTP request header in JAX-RS (2 ways)?**

* inject directly with @HeaderParam;

**import** **javax.ws.rs.GET**;

**import** **javax.ws.rs.Path**;

**import** **javax.ws.rs.HeaderParam**;

**import** **javax.ws.rs.core.Response**;

**@Path**("/persons")

public **class** **PersonService** {

**@GET**

**@Path**("/get")

public Response getPerson(

**@HeaderParam**("person-agent") String personAgent) {

**return** Response.status(**200**)

.entity("getPerson is called, personAgent : " + personAgent)

.build();

}

}

On calling URI: “/persons/get” result: getPerson is called, personAgent : Mozilla**/**5.0 **(**Windows NT 6.1; rv:5.0**)** Gecko**/**20100101 Firefox**/**5.0

* pragmatically via @Context.

**import** **javax.ws.rs.GET**;

**import** **javax.ws.rs.Path**;

**import** **javax.ws.rs.core.Context**;

**import** **javax.ws.rs.core.HttpHeaders**;

**import** **javax.ws.rs.core.Response**;

**@Path**("/persons")

public **class** **PersonService** {

**@GET**

**@Path**("/get")

public Response getPerson(**@Context** HttpHeaders headers) {

String personAgent = headers.getRequestHeader("person-agent").get(**0**);

**return** Response.status(**200**)

.entity("getPerson is called, personAgent : " + personAgent)

.build();

}

}

On calling URI: “/persons/get” result: getPerson is called, personAgent : Mozilla**/**5.0 **(**Windows NT 6.1; rv:5.0**)** Gecko**/**20100101 Firefox**/**5.0

**Q40  How to download file in JAX-RS?**

* put @Produces(“?”) on service method, with a Response return type. Instead “?” write a type text/plain, image/png, etc.
* set “Content-Disposition” in Response header to tell browser pop up a download box for user to download.

**import** **java.io.File**;

**import** **javax.ws.rs.GET**;

**import** **javax.ws.rs.Path**;

**import** **javax.ws.rs.Produces**;

**import** **javax.ws.rs.core.Response**;

**import** **javax.ws.rs.core.Response.ResponseBuilder**;

**@Path**("/image")

public **class** **ImageService** {

private static final String FILE\_PATH = "c:**\\**my.png";

**@GET**

**@Path**("/get")

**@Produces**("image/png")

public Response getFile() {

File file = new File(FILE\_PATH);

ResponseBuilder response = Response.ok((Object) file);

response.header("Content-Disposition",

"attachment; filename=image\_from\_server.png");

**return** response.build();

}

}

# Spring

## Introduction

Spring is an open-source framework created to address the complexity of an enterprise application development

* One of the chief advantages of the Spring framework is its layered architecture, which allows the developer to be selective about which of its components they can use while providing a cohesive framework for J2EE application development
* Spring framework provides support and integration to various technologies for e.g.:
  + - Support for Transaction Management
    - Support for interaction with the different databases
    - Integration with the Object Relationship frameworks for e.g. Hibernate, iBatis etc
    - Support for Dependency Injection which means all the required dependencies will be resolved with the help of containers
    - Support for REST style web-services

## Introduction to the Spring IoC container and beans

<https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html>

This chapter covers the Spring Framework implementation of the Inversion of Control (IoC) [[1]](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html" \l "ftn.d5e1144)principle. IoC is also known as dependency injection (DI). It is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the Service Locator pattern

The org.springframework.beans and org.springframework.context packages are the basis for Spring Framework's IoC container. The [BeanFactory](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/BeanFactory.html) interface provides an advanced configuration mechanism capable of managing any type of object. [ApplicationContext](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/context/ApplicationContext.html) is a sub-interface of BeanFactory. It adds easier integration with Spring's AOP features; message resource handling (for use in internationalization), event publication; and application-layer specific contexts such as the WebApplicationContext for use in web applications.

In short, the BeanFactory provides the configuration framework and basic functionality, and the ApplicationContext adds more enterprise-specific functionality. The ApplicationContext is a complete superset of the BeanFactory, and is used exclusively in this chapter in descriptions of Spring's IoC container. For more information on using the BeanFactory instead of the ApplicationContext,

## 5.2 Container overview

The **interface** org.springframework.context.ApplicationContext **represents the Spring IoC container and is responsible for** **instantiating, configuring, and assembling the aforementioned beans**. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration **metadata**. The configuration metadata is represented in **XML, Java annotations, or Java code**. It allows you to express the objects that compose your application and the rich interdependencies between such objects.

------------------------------------------------------------------------------

Spring framework is an open source Java platform that provides comprehensive infrastructure support for developing robust Java applications very easily and very rapidly.

Spring framework is an open source Java platform. Spring is the most popular application development framework for enterprise Java. Millions of developers around the world use Spring Framework to create high performing, easily testable, and reusable code.

Spring framework targets to make J2EE development easier to use and promotes good programming practices by enabling a POJO-based programming model.

## Following is the list of few of the great benefits of using Spring Framework −

* **POJO Based** - Spring enables developers to develop enterprise-class applications using POJOs. The benefit of using only POJOs is that you do not need an EJB container product such as an application server but you have the option of using only a robust servlet container such as Tomcat or some commercial product

**Testablity** - Testing an application written with Spring is simple because environment-dependent code is moved into this framework. Furthermore, by using JavaBeanstyle POJOs, it becomes easier to use dependency injection for injecting test data.

* **Web MVC** - Spring's web framework is a well-designed web MVC framework, which provides a great alternative to web frameworks such as Struts or other over-engineered or less popular web frameworks.
* **Central Exception Handling** - Spring provides a convenient API to translate technology-specific exceptions (thrown by JDBC, Hibernate, or JDO, for example) into consistent, unchecked exceptions.
* **Lightweight** - Lightweight IoC containers tend to be lightweight, especially when compared to EJB containers, for example. This is beneficial for developing and deploying applications on computers with limited memory and CPU resources.
* **Transaction management** - Spring provides a consistent transaction management interface that can scale down to a local transaction (using a single database, for example) and scale up to global transactions (using JTA, for example).

## 5.15 The BeanFactory

The BeanFactory provides the underlying basis for Spring's IoC functionality but it is only used directly in integration with other third-party frameworks and is now largely historical in nature for most users of Spring. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purposes of backward compatibility with the large number of third-party frameworks that integrate with Spring. Often third-party components that can not use more modern equivalents such as @PostConstruct or @PreDestroy in order to remain compatible with JDK 1.4 or to avoid a dependency on JSR-250.

This section provides additional background into the differences between the BeanFactory and ApplicationContext and how one might access the IoC container directly through a classic singleton lookup.

### 5.15.1 BeanFactory or ApplicationContext?

Use an ApplicationContext unless you have a good reason for not doing so.

Because the ApplicationContext includes all functionality of the BeanFactory, it is generally recommended over the BeanFactory, except for a few situations such as in an Applet where memory consumption might be critical and a few extra kilobytes might make a difference. However, for most typical enterprise applications and systems, the ApplicationContext is what you will want to use. Spring 2.0 and later makes heavy use of the [BeanPostProcessor extension point](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-extension-bpp) (to effect proxying and so on). If you use only a plain BeanFactory, a fair amount of support such as transactions and AOP will not take effect, at least not without some extra steps on your part. This situation could be confusing because nothing is actually wrong with the configuration.

The following table lists features provided by the BeanFactory and ApplicationContext interfaces and implementations.

**Table 5.8. Feature Matrix**

| **Feature** | **BeanFactory** | **ApplicationContext** |
| --- | --- | --- |
| Bean instantiation/wiring | Yes | Yes |
| Automatic BeanPostProcessor registration | No | Yes |
| Automatic BeanFactoryPostProcessor registration | No | Yes |
| Convenient MessageSource access (for i18n) | No | Yes |
| ApplicationEvent publication | No | Yes |

## Dependency Injection (DI)

The technology that Spring is most identified with is the **Dependency Injection (DI) flavor of Inversion of Control**. The **Inversion of Control (IoC)** is a general concept, and it can be expressed in many different ways. **Dependency Injection is merely one concrete example of Inversion of Control.**

When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while unit testing. Dependency Injection helps in gluing these classes together and at the same time keeping them independent.

What is dependency injection exactly? Let's look at these two words separately. Here the dependency part translates into an association between two classes. For example, class A is dependent of class B. Now, let's look at the second part, injection. All this means is, class B will get injected into class A by the IoC.

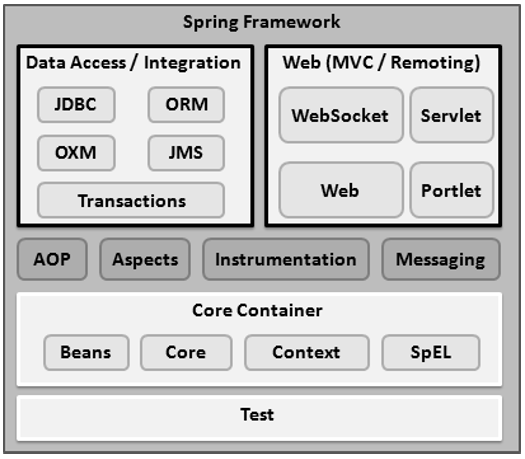
Dependency injection can happen in the way of passing parameters to the constructor or by post-construction using setter methods. As Dependency Injection is the heart of Spring Framework, we will explain this concept in a separate chapter with relevant example.

**(application class**" doesn't have a specific meaning in **Java** programming. Generally, it means the entry point into the **application**. In a basic command-line **application**, it is whatever **class** contains the public static void main(String[] args) method used to launch the program) )

## Modules

* Spring could potentially be a one-stop shop for all your enterprise applications. However, Spring is modular, allowing you to pick and choose which modules are applicable to you, without having to bring in the rest. The following section provides details about all the modules available in Spring Framework.

The Spring Framework provides about 20 modules which can be used based on an application requirement.



### Core Container

The Core Container consists of the Core, Beans, Context, and Expression Language modules the details of which are as follows −

* The **Core** module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.
* The **Bean** module provides BeanFactory, which is a sophisticated implementation of the factory pattern.
* The **Context** module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The ApplicationContext interface is the focal point of the Context module.
* The **SpEL** module provides a powerful expression language for querying and manipulating an object graph at runtime.

### Data Access/Integration

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules whose detail is as follows −

* The **JDBC** module provides a JDBC-abstraction layer that removes the need for tedious JDBC related coding.
* The **ORM** module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
* The **OXM** module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
* The Java Messaging Service **JMS** module contains features for producing and consuming messages.
* The **Transaction** module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

### Web

The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules the details of which are as follows −

* The **Web** module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
* The **Web-MVC** module contains Spring's Model-View-Controller (MVC) implementation for web applications.
* The **Web-Socket** module provides support for WebSocket-based, two-way communication between the client and the server in web applications.
* The **Web-Portlet** module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

### Miscellaneous

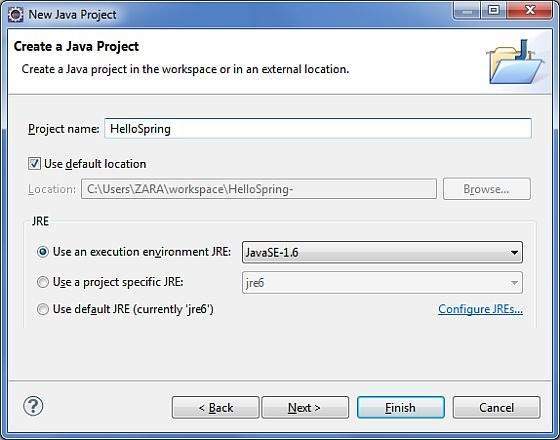
There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules the details of which are as follows −

* The **AOP** module provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
* The **Aspects** module provides integration with AspectJ, which is again a powerful and mature AOP framework.
* The **Instrumentation** module provides class instrumentation support and class loader implementations to be used in certain application servers.
* The **Messaging** module provides support for STOMP as the WebSocket sub-protocol to use in applications. It also supports an annotation programming model for routing and processing STOMP messages from WebSocket clients.
* The **Test** module supports the testing of Spring components with JUnit or TestNG frameworks.

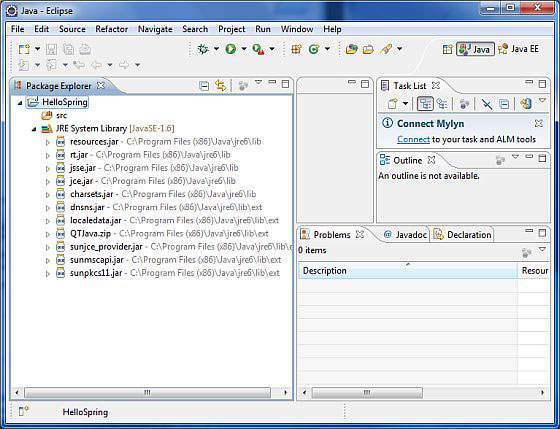
## Spring - Hello World Example

### Step 1 - Create Java Project

The first step is to create a simple Java Project using Eclipse IDE. Follow the option **File → New → Project** and finally select **Java Project** wizard from the wizard list. Now name your project as **HelloSpring** using the wizard window as follows −

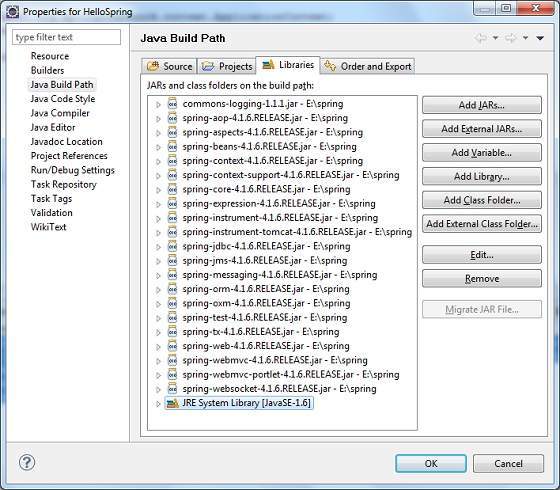


Once your project is created successfully, you will have the following content in your **Project Explorer** −



### Step 2 - Add Required Libraries

As a second step let us add Spring Framework and common logging API libraries in our project. To do this, right-click on your project name **HelloSpring** and then follow the following option available in the context menu − **Build Path → Configure Build Path** to display the Java Build Path window as follows −



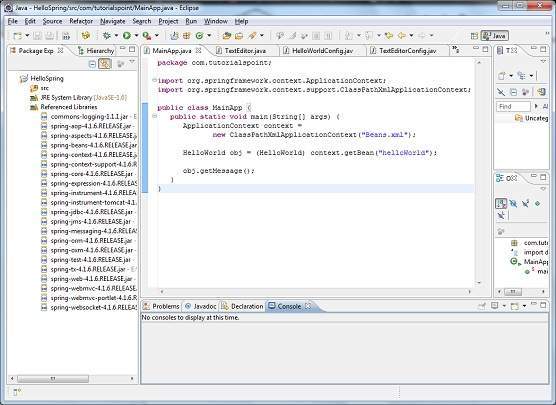
Now use **Add External JARs** button available under the **Libraries** tab to add the following core JARs from Spring Framework and Common Logging installation directories −

* commons-logging-1.1.1
* spring-aop-4.1.6.RELEASE
* spring-aspects-4.1.6.RELEASE
* spring-beans-4.1.6.RELEASE
* spring-context-4.1.6.RELEASE
* spring-context-support-4.1.6.RELEASE
* spring-core-4.1.6.RELEASE
* spring-expression-4.1.6.RELEASE
* spring-instrument-4.1.6.RELEASE
* spring-instrument-tomcat-4.1.6.RELEASE
* spring-jdbc-4.1.6.RELEASE
* spring-jms-4.1.6.RELEASE
* spring-messaging-4.1.6.RELEASE
* spring-orm-4.1.6.RELEASE
* spring-oxm-4.1.6.RELEASE
* spring-test-4.1.6.RELEASE
* spring-tx-4.1.6.RELEASE
* spring-web-4.1.6.RELEASE
* spring-webmvc-4.1.6.RELEASE
* spring-webmvc-portlet-4.1.6.RELEASE
* spring-websocket-4.1.6.RELEASE

### Step 3 - Create Source Files

Now let us create actual source files under the **HelloSpring** project. First we need to create a package called **com.tutorialspoint**. To do this, right click on **src** in package explorer section and follow the option − **New → Package**.

Next we will create **HelloWorld.java** and **MainApp.java** files under the com.tutorialspoint package.



Here is the content of **HelloWorld.java** file −

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}

Following is the content of the second file **MainApp.java** −

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

**ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");**

**HelloWorld obj = (HelloWorld) context.getBean("helloWorld");**

obj.getMessage();

}

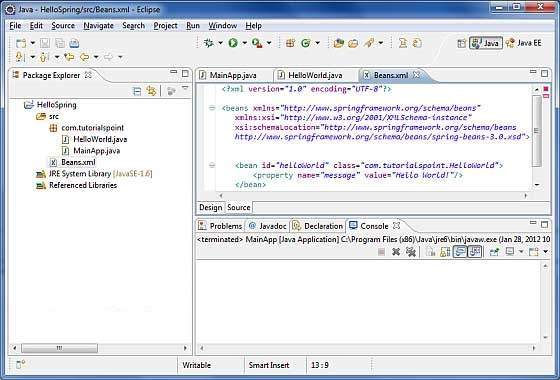
}

Following two important points are to be noted about the main program −

* The first step is to **create an application context** where we used framework API **ClassPathXmlApplicationContext()**. **This API loads beans configuration file and eventually based on the provided API, it takes care of creating and initializing all the objects, i.e. beans mentioned in the configuration file**.
* The second step is used to **get the required bean using getBean()** method of the **created** context. This method uses bean ID to return a generic object, which finally can be casted to the actual object (**by casting to the class type ie (HelloWorld) here** ). Once you have an object, you can use this object to call any class method.

### Step 4 - Create Bean Configuration File

You need to **create a Bean Configuration file which is an XML file and acts as a cement that glues the beans, i.e. the classes together**. This file needs to be created under the **src** directory as shown in the following screenshot −



Usually developers name this file as **Beans.xml, but you are independent to choose any name** you like. You have to make sure that this file is **available in CLASSPATH** and use the **same name in the main application** while creating an application context as shown in MainApp.java file.

The **Beans.xml is used to assign unique IDs to different beans and to control the creation of objects with different values without impacting any of the Spring source files**. For example, using the following file you can pass any value for "message" variable and you can print different values of message without impacting HelloWorld.java and MainApp.java files. Let us see how it works −

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld">

<property name = "message" value = "Hello World!"/>

</bean>

</beans>

When Spring application gets loaded into the memory, Framework makes use of the above configuration file to create all the beans defined and assigns them a unique ID as defined in **<bean>** tag. You can use **<property>** tag to pass the values of different variables used at the time of object creation.

### Step 5 - Running the Program

Once you are done with creating the source and beans configuration files, you are ready for this step, which is compiling and running your program. To do this, keep MainApp.Java file tab active and use either **Run** option available in the Eclipse IDE or use **Ctrl + F11** to compile and run your **MainApp** application. If everything is fine with your application, this will print the following message in Eclipse IDE's console −

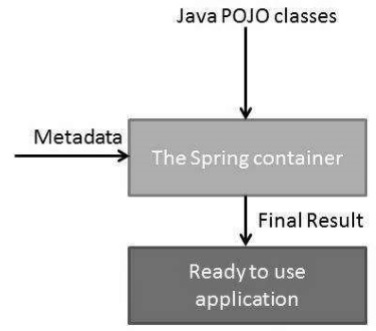
Your Message : Hello World!

Congratulations, you have successfully created your first Spring Application. You can see the flexibility of the above Spring application by changing the value of "message" property and keeping both the source files unchanged.

## Spring - IoC Containers

The Spring container is at the core of the Spring Framework. The **container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction.** The **Spring container uses DI** to manage the components that make up an application. These **objects are called Spring Beans**, which we will discuss in the next chapter.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram represents a high-level view of how Spring works. **The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.**



Spring provides the following two distinct types of containers.

|  |  |
| --- | --- |
| **Sr.No.** | **Container & Description** |
| 1 | [Spring BeanFactory Container](https://www.tutorialspoint.com/spring/spring_beanfactory_container.htm)  This is the **simplest container providing the basic support for DI** and is defined by the *org.springframework.beans.factory.BeanFactory* interface. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purpose of backward compatibility with a large number of third-party frameworks that integrate with Spring. |
| 2 | [Spring ApplicationContext Container](https://www.tutorialspoint.com/spring/spring_applicationcontext_container.htm)  This container **adds more enterprise-specific functionality** such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the *org.springframework.context.ApplicationContext* interface. |

**The *ApplicationContext* container includes all functionality of the *BeanFactory*container,** so it is generally recommended over *BeanFactory*. BeanFactory can still be used for lightweight applications like mobile devices or applet-based applications where data volume and speed is significant.

## Spring - Bean Definition

The objects that form the backbone of your application and that are managed by the Spring IoC container are called **beans**. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container. For example, in the form of XML <bean/> definitions which you have already seen in the previous chapters.

Bean definition contains the information called **configuration metadata**, which is needed for the container to know the following −

* How to create a bean
* Bean's lifecycle details
* Bean's dependencies

All the above configuration me

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- A simple bean definition -->

<bean id = "..." class = "...">

<!-- collaborators and configuration for this bean go here -->

</bean>

<!-- A bean definition with lazy init set on -->

<bean id = "..." class = "..." lazy-init = "true">

<!-- collaborators and configuration for this bean go here -->

</bean>

<!-- A bean definition with initialization method -->

<bean id = "..." class = "..." init-method = "...">

<!-- collaborators and configuration for this bean go here -->

</bean>

<!-- A bean definition with destruction method -->

<bean id = "..." class = "..." destroy-method = "...">

<!-- collaborators and configuration for this bean go here -->

</bean>

<!-- more bean definitions go here -->

</beans>

## Spring - Bean Scopes

When defining a <bean> you have the option of declaring a scope for that bean. For example, to force Spring to produce a new bean instance each time one is needed, you should declare the bean's scope attribute to be **prototype**. Similarly, if you want Spring to return the same bean instance each time one is needed, you should declare the bean's scope attribute to be **singleton**.

The Spring Framework supports the following five scopes, three of which are available only if you use a web-aware ApplicationContext.

|  |  |
| --- | --- |
| **Sr.No.** | **Scope & Description** |
| 1 | **singleton**  This scopes the bean definition to a single instance per Spring IoC container (default). |
| 2 | **prototype**  This scopes a single bean definition to have any number of object instances. |
| 3 | **request**  This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext. |
| 4 | **session**  This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |
| 5 | **global-session**  This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |

### The singleton scope

If a scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.

The default scope is always singleton. However, when you need one and only one instance of a bean, you can set the **scope**property to **singleton** in the bean configuration file, as shown in the following code snippet −

<!-- A bean definition with singleton scope -->

<bean id = "..." class = "..." scope = "singleton">

<!-- collaborators and configuration for this bean go here -->

</bean>

Example

Let us have a working Eclipse IDE in place and take the following steps to create a Spring application −

|  |  |
| --- | --- |
| **Steps** | **Description** |
| 1 | Create a project with a name *SpringExample* and create a package *com.tutorialspoint* under the **src** folder in the created project. |
| 2 | Add required Spring libraries using *Add External JARs* option as explained in the *Spring Hello World Example* chapter. |
| 3 | Create Java classes *HelloWorld* and *MainApp* under the *com.tutorialspoint* package. |
| 4 | Create Beans configuration file *Beans.xml* under the **src** folder. |
| 5 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Here is the content of **HelloWorld.java** file −

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}

Following is the content of the **MainApp.java** file −

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

HelloWorld objA = (HelloWorld) context.getBean("helloWorld");

objA.setMessage("I'm object A");

objA.getMessage();

HelloWorld objB = (HelloWorld) context.getBean("helloWorld");

objB.getMessage();

}

}

Following is the configuration file **Beans.xml** required for singleton scope −

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld" scope = "singleton">

</bean>

</beans>

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message −

Your Message : I'm object A

Your Message : I'm object A

### The prototype scope

If the scope is set to prototype, the Spring IoC container creates a new bean instance of the object every time a request for that specific bean is made. As a rule, use the prototype scope for all state-full beans and the singleton scope for stateless beans.

To define a prototype scope, you can set the **scope** property to **prototype** in the bean configuration file, as shown in the following code snippet −

<!-- A bean definition with prototype scope -->

<bean id = "..." class = "..." scope = "prototype">

<!-- collaborators and configuration for this bean go here -->

</bean>

Here is the content of **HelloWorld.java** file

package com.tutorialspoint;

public class HelloWorld {

private String message;

public void setMessage(String message){

this.message = message;

}

public void getMessage(){

System.out.println("Your Message : " + message);

}

}

Following is the content of the **MainApp.java** file −

package com.tutorialspoint;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

HelloWorld objA = (HelloWorld) context.getBean("helloWorld");

objA.setMessage("I'm object A");

objA.getMessage();

HelloWorld objB = (HelloWorld) context.getBean("helloWorld");

objB.getMessage();

}

}

Following is the configuration file **Beans.xml** required for prototype scope −

<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld" scope = "prototype">

</bean>

</beans>

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message −

Your Message : I'm object A

Your Message : null

## Spring - Bean Life Cycle

### The life cycle of a Spring bean is easy to understand. When a bean is instantiated, it may be required to perform some initialization to get it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required.

To define setup and teardown for a bean, we simply declare the <bean> with **initmethod** and/or **destroy-method** parameters. The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, destroymethod specifies a method that is called just before a bean is removed from the container.

### 5.2.2 Instantiating a container

Instantiating a Spring IoC container is straightforward. The location path or paths supplied to an ApplicationContext constructor are actually resource strings that allow the container to load configuration metadata from a variety of external resources such as the local file system, from the Java CLASSPATH, and so on.

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {"services.xml", "daos.xml"});

|  |
| --- |
| [Note] |
| After you learn about Spring's IoC container, you may want to know more about Spring's Resource abstraction, as described in [Chapter 6, *Resources*](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/resources.html), which provides a convenient mechanism for reading an InputStream from locations defined in a URI syntax. In particular, Resource paths are used to construct applications contexts as described in [Section 6.7, “Application contexts and Resource paths”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/resources.html#resources-app-ctx). |

The following example shows the service layer objects (services.xml) configuration file:

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

*<!-- services -->*

<bean id="petStore"

class="org.springframework.samples.jpetstore.services.PetStoreServiceImpl">

<property name="accountDao" ref="accountDao"/>

<property name="itemDao" ref="itemDao"/>

*<!-- additional collaborators and configuration for this bean go here -->*

</bean>

*<!-- more bean definitions for services go here -->*

</beans>

The following example shows the data access objects daos.xml file:

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="accountDao"

class="org.springframework.samples.jpetstore.dao.ibatis.SqlMapAccountDao">

*<!-- additional collaborators and configuration for this bean go here -->*

</bean>

<bean id="itemDao" class="org.springframework.samples.jpetstore.dao.ibatis.SqlMapItemDao">

*<!-- additional collaborators and configuration for this bean go here -->*

</bean>

*<!-- more bean definitions for data access objects go here -->*

</beans>

In the preceding example, the service layer consists of the class PetStoreServiceImpl, and two data access objects of the type SqlMapAccountDao and SqlMapItemDao are based on the [iBatis](https://ibatis.apache.org/) Object/Relational mapping framework. The property name element refers to the name of the JavaBean property, and the ref element refers to the name of another bean definition. This linkage between id and ref elements expresses the dependency between collaborating objects. For details of configuring an object's dependencies, see [Dependencies](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-dependencies).

#### Composing XML-based configuration metadata

It can be useful to have bean definitions span multiple XML files. Often each individual XML configuration file represents a logical layer or module in your architecture.

You can use the **application context constructor** to load bean definitions from all these XML fragments. This constructor takes multiple Resource locations, as was shown in the previous section. Alternatively, use one or more occurrences of the <import/> element to load bean definitions from another file or files. For example:

<beans>

<import resource="services.xml"/>

<import resource="resources/messageSource.xml"/>

<import resource="/resources/themeSource.xml"/>

<bean id="bean1" class="..."/>

<bean id="bean2" class="..."/>

</beans>

In the preceding example, external bean definitions are loaded from three files, services.xml, messageSource.xml, and themeSource.xml. All location paths are relative to the definition file doing the importing, so services.xml must be in the same directory or classpath location as the file doing the importing, while messageSource.xml and themeSource.xml must be in a resources location below the location of the importing file. As you can see, a leading slash is ignored, but given that these paths are relative, it is better form not to use the slash at all. The contents of the files being imported, including the top level <beans/> element, must be valid XML bean definitions according to the Spring Schema or DTD.

|  |
| --- |
| [Note] |
| It is possible, but not recommended, to reference files in parent directories using a relative "../" path. Doing so creates a dependency on a file that is outside the current application. In particular, this reference is not recommended for "classpath:" URLs (for example, "classpath:../services.xml"), where the runtime resolution process chooses the "nearest" classpath root and then looks into its parent directory. Classpath configuration changes may lead to the choice of a different, incorrect directory.  You can always use fully qualified resource locations instead of relative paths: for example, "file:C:/config/services.xml" or "classpath:/config/services.xml". However, be aware that you are coupling your application's configuration to specific absolute locations. It is generally preferable to keep an indirection for such absolute locations, for example, through "${...}" placeholders that are resolved against JVM system properties at runtime. |

|  |  |  |
| --- | --- | --- |
| **5. The IoC container** | | |
| [Prev](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/spring-core.html) | **Part III. Core Technologies** | [Next](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/resources.html) |

## 5. The IoC container

## 5.1 Introduction to the Spring IoC container and beans

This chapter covers the Spring Framework implementation of the Inversion of Control (IoC) [[1]](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#ftn.d5e1144)principle. IoC is also known as dependency injection (DI). It is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the Service Locator pattern.

The org.springframework.beans and org.springframework.context packages are the basis for Spring Framework's IoC container. The [BeanFactory](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/BeanFactory.html) interface provides an advanced configuration mechanism capable of managing any type of object. [ApplicationContext](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/context/ApplicationContext.html) is a sub-interface of BeanFactory. It adds easier integration with Spring's AOP features; message resource handling (for use in internationalization), event publication; and application-layer specific contexts such as the WebApplicationContext for use in web applications.

In short, the BeanFactory provides the configuration framework and basic functionality, and the ApplicationContext adds more enterprise-specific functionality. The ApplicationContext is a complete superset of the BeanFactory, and is used exclusively in this chapter in descriptions of Spring's IoC container. For more information on using the BeanFactory instead of the ApplicationContext, refer to [Section 5.15, “The BeanFactory”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-beanfactory).

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.

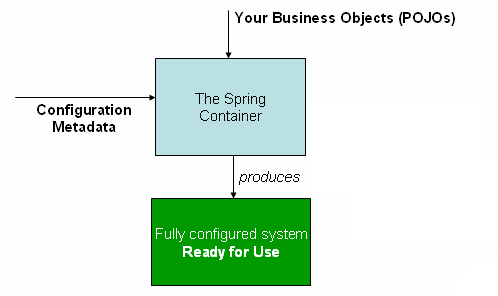
## 5.2 Container overview

The interface org.springframework.context.ApplicationContext represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the aforementioned beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. It allows you to express the objects that compose your application and the rich interdependencies between such objects.

Several implementations of the ApplicationContext interface are supplied out-of-the-box with Spring. In standalone applications it is common to create an instance of [ClassPathXmlApplicationContext](http://static.springsource.org/spring/docs/current/api/org/springframework/context/support/ClassPathXmlApplicationContext.html) or [FileSystemXmlApplicationContext](http://static.springsource.org/spring/docs/current/api/org/springframework/context/support/FileSystemXmlApplicationContext.html). While XML has been the traditional format for defining configuration metadata you can instruct the container to use Java annotations or code as the metadata format by providing a small amount of XML configuration to declaratively enable support for these additional metadata formats.

In most application scenarios, explicit user code is not required to instantiate one or more instances of a Spring IoC container. For example, in a web application scenario, a simple eight (or so) lines of boilerplate J2EE web descriptor XML in the web.xml file of the application will typically suffice (see [Section 5.14.4, “Convenient ApplicationContext instantiation for web applications”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#context-create)). If you are using the [SpringSource Tool Suite](http://www.springsource.com/produts/sts) Eclipse-powered development environment or [Spring Roo](http://www.springsource.org/roo) this boilerplate configuration can be easily created with few mouse clicks or keystrokes.

The following diagram is a high-level view of how Spring works. Your application classes are combined with configuration metadata so that after the ApplicationContext is created and initialized, you have a fully configured and executable system or application.



The Spring IoC container

### 5.2.1 Configuration metadata

As the preceding diagram shows, the Spring IoC container consumes a form of configuration metadata; this configuration metadata represents how you as an application developer tell the Spring container to instantiate, configure, and assemble the objects in your application.

Configuration metadata is traditionally supplied in a simple and intuitive XML format, which is what most of this chapter uses to convey key concepts and features of the Spring IoC container.

|  |
| --- |
| [Note] |
| XML-based metadata is not the only allowed form of configuration metadata. The Spring IoC container itself is totally decoupled from the format in which this configuration metadata is actually written. |

For information about using other forms of metadata with the Spring container, see:

* [Annotation-based configuration](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-annotation-config): Spring 2.5 introduced support for annotation-based configuration metadata.
* [Java-based configuration](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-java): Starting with Spring 3.0, many features provided by the [Spring JavaConfig project](http://www.springsource.org/javaconfig) became part of the core Spring Framework. Thus you can define beans external to your application classes by using Java rather than XML files. To use these new features, see the @Configuration, @Bean, @Import and @DependsOn annotations.

Spring configuration consists of at least one and typically more than one bean definition that the container must manage. XML-based configuration metadata shows these beans configured as <bean/> elements inside a top-level <beans/> element.

These bean definitions correspond to the actual objects that make up your application. Typically you define service layer objects, data access objects (DAOs), presentation objects such as Struts Action instances, infrastructure objects such as Hibernate SessionFactories, JMS Queues, and so forth. Typically one does not configure fine-grained domain objects in the container, because it is usually the responsibility of DAOs and business logic to create and load domain objects. However, you can use Spring's integration with AspectJ to configure objects that have been created outside the control of an IoC container. See [Using AspectJ to dependency-inject domain objects with Spring](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/aop.html#aop-atconfigurable).

The following example shows the basic structure of XML-based configuration metadata:

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="..." class="...">

*<!-- collaborators and configuration for this bean go here -->*

</bean>

<bean id="..." class="...">

*<!-- collaborators and configuration for this bean go here -->*

</bean>

*<!-- more bean definitions go here -->*

</beans>

The id attribute is a string that you use to identify the individual bean definition. The class attribute defines the type of the bean and uses the fully qualified classname. The value of the id attribute refers to collaborating objects. The XML for referring to collaborating objects is not shown in this example; see [Dependencies](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-dependencies) for more information.

### 5.2.2 Instantiating a container

Instantiating a Spring IoC container is straightforward. The location path or paths supplied to an ApplicationContext constructor are actually resource strings that allow the container to load configuration metadata from a variety of external resources such as the local file system, from the Java CLASSPATH, and so on.

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {"services.xml", "daos.xml"});

|  |
| --- |
| [Note] |
| After you learn about Spring's IoC container, you may want to know more about Spring's Resource abstraction, as described in [Chapter 6, *Resources*](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/resources.html), which provides a convenient mechanism for reading an InputStream from locations defined in a URI syntax. In particular, Resource paths are used to construct applications contexts as described in [Section 6.7, “Application contexts and Resource paths”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/resources.html#resources-app-ctx). |

The following example shows the service layer objects (services.xml) configuration file:

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

*<!-- services -->*

<bean id="petStore"

class="org.springframework.samples.jpetstore.services.PetStoreServiceImpl">

<property name="accountDao" ref="accountDao"/>

<property name="itemDao" ref="itemDao"/>

*<!-- additional collaborators and configuration for this bean go here -->*

</bean>

*<!-- more bean definitions for services go here -->*

</beans>

The following example shows the data access objects daos.xml file:

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="accountDao"

class="org.springframework.samples.jpetstore.dao.ibatis.SqlMapAccountDao">

*<!-- additional collaborators and configuration for this bean go here -->*

</bean>

<bean id="itemDao" class="org.springframework.samples.jpetstore.dao.ibatis.SqlMapItemDao">

*<!-- additional collaborators and configuration for this bean go here -->*

</bean>

*<!-- more bean definitions for data access objects go here -->*

</beans>

In the preceding example, the service layer consists of the class PetStoreServiceImpl, and two data access objects of the type SqlMapAccountDao and SqlMapItemDao are based on the [iBatis](https://ibatis.apache.org/) Object/Relational mapping framework. The property name element refers to the name of the JavaBean property, and the ref element refers to the name of another bean definition. This linkage between id and ref elements expresses the dependency between collaborating objects. For details of configuring an object's dependencies, see [Dependencies](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-dependencies).

#### Composing XML-based configuration metadata

It can be useful to have bean definitions span multiple XML files. Often each individual XML configuration file represents a logical layer or module in your architecture.

You can use the application context constructor to load bean definitions from all these XML fragments. This constructor takes multiple Resource locations, as was shown in the previous section. Alternatively, use one or more occurrences of the <import/> element to load bean definitions from another file or files. For example:

<beans>

<import resource="services.xml"/>

<import resource="resources/messageSource.xml"/>

<import resource="/resources/themeSource.xml"/>

<bean id="bean1" class="..."/>

<bean id="bean2" class="..."/>

</beans>

In the preceding example, external bean definitions are loaded from three files, services.xml, messageSource.xml, and themeSource.xml. All location paths are relative to the definition file doing the importing, so services.xml must be in the same directory or classpath location as the file doing the importing, while messageSource.xml and themeSource.xml must be in a resources location below the location of the importing file. As you can see, a leading slash is ignored, but given that these paths are relative, it is better form not to use the slash at all. The contents of the files being imported, including the top level <beans/> element, must be valid XML bean definitions according to the Spring Schema or DTD.

|  |
| --- |
| [Note] |
| It is possible, but not recommended, to reference files in parent directories using a relative "../" path. Doing so creates a dependency on a file that is outside the current application. In particular, this reference is not recommended for "classpath:" URLs (for example, "classpath:../services.xml"), where the runtime resolution process chooses the "nearest" classpath root and then looks into its parent directory. Classpath configuration changes may lead to the choice of a different, incorrect directory.  You can always use fully qualified resource locations instead of relative paths: for example, "file:C:/config/services.xml" or "classpath:/config/services.xml". However, be aware that you are coupling your application's configuration to specific absolute locations. It is generally preferable to keep an indirection for such absolute locations, for example, through "${...}" placeholders that are resolved against JVM system properties at runtime. |

### 5.2.3 Using the container

The ApplicationContext is the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies. Using the method T getBean(String name, Class<T> requiredType) you can retrieve instances of your beans.

The ApplicationContext enables you to read bean definitions and access them as follows:

*// create and configure beans*

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[] {"services.xml", "daos.xml"});

*// retrieve configured instance*

PetStoreServiceImpl service = context.getBean("petStore", PetStoreServiceImpl.**class**);

*// use configured instance*

List userList = service.getUsernameList();

You use getBean() to retrieve instances of your beans. The ApplicationContext interface has a few other methods for retrieving beans, but ideally your application code should never use them. Indeed, your application code should have no calls to the getBean() method at all, and thus no dependency on Spring APIs at all. For example, Spring's integration with web frameworks provides for dependency injection for various web framework classes such as controllers and JSF-managed beans.

## 5.3 Bean overview

A Spring IoC container manages one or more beans. These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML <bean/> definitions.

Within the container itself, these bean definitions are represented as BeanDefinition objects, which contain (among other information) the following metadata:

* A package-qualified class name: typically the actual implementation class of the bean being defined.
* Bean behavioral configuration elements, which state how the bean should behave in the container (scope, lifecycle callbacks, and so forth).
* References to other beans that are needed for the bean to do its work; these references are also called collaborators or dependencies.
* Other configuration settings to set in the newly created object, for example, the number of connections to use in a bean that manages a connection pool, or the size limit of the pool.

This metadata translates to a set of properties that make up each bean definition.

**Table 5.1. The bean definition**

| **Property** | **Explained in...** |
| --- | --- |
| class | [Section 5.3.2, “Instantiating beans”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-class) |
| name | [Section 5.3.1, “Naming beans”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-beanname) |
| scope | [Section 5.5, “Bean scopes”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes) |
| constructor arguments | [Section 5.4.1, “Dependency injection”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-collaborators) |
| properties | [Section 5.4.1, “Dependency injection”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-collaborators) |
| autowiring mode | [Section 5.4.5, “Autowiring collaborators”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-autowire) |
| lazy-initialization mode | [Section 5.4.4, “Lazy-initialized beans”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-lazy-init) |
| initialization method | [the section called “Initialization callbacks”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-lifecycle-initializingbean) |
| destruction method | [the section called “Destruction callbacks”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-lifecycle-disposablebean) |

In addition to bean definitions that contain information on how to create a specific bean, the ApplicationContext implementations also permit the registration of existing objects that are created outside the container, by users. This is done by accessing the ApplicationContext's BeanFactory via the method getBeanFactory() which returns the BeanFactory implementation DefaultListableBeanFactory. DefaultListableBeanFactory supports this registration through the methods registerSingleton(..) and registerBeanDefinition(..). However, typical applications work solely with beans defined through metadata bean definitions.

### 5.3.1 Naming beans

Every bean has one or more identifiers. These identifiers must be unique within the container that hosts the bean. A bean usually has only one identifier, but if it requires more than one, the extra ones can be considered aliases.

In XML-based configuration metadata, you use the id and/or name attributes to specify the bean identifier(s). The id attribute allows you to specify exactly one id. Conventionally these names are alphanumeric ('myBean', 'fooService', etc), but may special characters as well. If you want to introduce other aliases to the bean, you can also specify them in the name attribute, separated by a comma (,), semicolon (;), or white space. As a historical note, in versions prior to Spring 3.1, the id attribute was typed as an xsd:ID, which constrained possible characters. As of 3.1, it is now xsd:string. Note that bean id uniqueness is still enforced by the container, though no longer by XML parsers.

You are not required to supply a name or id for a bean. If no name or id is supplied explicitly, the container generates a unique name for that bean. However, if you want to refer to that bean by name, through the use of the ref element or [Service Locator](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-servicelocator) style lookup, you must provide a name. Motivations for not supplying a name are related to using [inner beans](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-inner-beans) and [autowiring collaborators](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-autowire).

**Bean naming conventions**

The convention is to use the standard Java convention for instance field names when naming beans. That is, bean names start with a lowercase letter, and are camel-cased from then on. Examples of such names would be (without quotes) 'accountManager', 'accountService', 'userDao', 'loginController', and so forth.

Naming beans consistently makes your configuration easier to read and understand, and if you are using Spring AOP it helps a lot when applying advice to a set of beans related by name.

#### Aliasing a bean outside the bean definition

In a bean definition itself, you can supply more than one name for the bean, by using a combination of up to one name specified by the id attribute, and any number of other names in the name attribute. These names can be equivalent aliases to the same bean, and are useful for some situations, such as allowing each component in an application to refer to a common dependency by using a bean name that is specific to that component itself.

Specifying all aliases where the bean is actually defined is not always adequate, however. It is sometimes desirable to introduce an alias for a bean that is defined elsewhere. This is commonly the case in large systems where configuration is split amongst each subsystem, each subsystem having its own set of object definitions. In XML-based configuration metadata, you can use the <alias/> element to accomplish this.

<alias name="fromName" alias="toName"/>

In this case, a bean in the same container which is named fromName, may also after the use of this alias definition, be referred to as toName.

For example, the configuration metadata for subsystem A may refer to a DataSource via the name 'subsystemA-dataSource. The configuration metadata for subsystem B may refer to a DataSource via the name 'subsystemB-dataSource'. When composing the main application that uses both these subsystems the main application refers to the DataSource via the name 'myApp-dataSource'. To have all three names refer to the same object you add to the MyApp configuration metadata the following aliases definitions:

<alias name="subsystemA-dataSource" alias="subsystemB-dataSource"/>

<alias name="subsystemA-dataSource" alias="myApp-dataSource" />

Now each component and the main application can refer to the dataSource through a name that is unique and guaranteed not to clash with any other definition (effectively creating a namespace), yet they refer to the same bean.

### 5.3.2 Instantiating beans

A bean definition essentially is a recipe for creating one or more objects. The container looks at the recipe for a named bean when asked, and uses the configuration metadata encapsulated by that bean definition to create (or acquire) an actual object.

If you use XML-based configuration metadata, you specify the type (or class) of object that is to be instantiated in the class attribute of the <bean/> element. This class attribute, which internally is a Class property on a BeanDefinition instance, is usually mandatory. (For exceptions, see [the section called “Instantiation using an instance factory method”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-class-instance-factory-method) and [Section 5.7, “Bean definition inheritance”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-child-bean-definitions).) You use the Class property in one of two ways:

* Typically, to specify the bean class to be constructed in the case where the container itself directly creates the bean by calling its constructor reflectively, somewhat equivalent to Java code using the new operator.
* To specify the actual class containing the static factory method that will be invoked to create the object, in the less common case where the container invokes a static, factory method on a class to create the bean. The object type returned from the invocation of the static factory method may be the same class or another class entirely.

**Inner class names**

If you want to configure a bean definition for a static nested class, you have to use the binary name of the inner class.

For example, if you have a class called Foo in the com.example package, and this Foo class has a static inner class called Bar, the value of the 'class' attribute on a bean definition would be...

com.example.Foo$Bar

Notice the use of the $ character in the name to separate the inner class name from the outer class name.

#### Instantiation with a constructor

When you create a bean by the constructor approach, all normal classes are usable by and compatible with Spring. That is, the class being developed does not need to implement any specific interfaces or to be coded in a specific fashion. Simply specifying the bean class should suffice. However, depending on what type of IoC you use for that specific bean, you may need a default (empty) constructor.

The Spring IoC container can manage virtually any class you want it to manage; it is not limited to managing true JavaBeans. Most Spring users prefer actual JavaBeans with only a default (no-argument) constructor and appropriate setters and getters modeled after the properties in the container. You can also have more exotic non-bean-style classes in your container. If, for example, you need to use a legacy connection pool that absolutely does not adhere to the JavaBean specification, Spring can manage it as well.

With XML-based configuration metadata you can specify your bean class as follows:

<bean id="exampleBean" class="examples.ExampleBean"/>

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>

For details about the mechanism for supplying arguments to the constructor (if required) and setting object instance properties after the object is constructed, see [Injecting Dependencies](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-collaborators).

#### Instantiation with a static factory method

When defining a bean that you create with a static factory method, you use the class attribute to specify the class containing the static factory method and an attribute named factory-method to specify the name of the factory method itself. You should be able to call this method (with optional arguments as described later) and return a live object, which subsequently is treated as if it had been created through a constructor. One use for such a bean definition is to call static factories in legacy code.

The following bean definition specifies that the bean will be created by calling a factory-method. The definition does not specify the type (class) of the returned object, only the class containing the factory method. In this example, the createInstance() method must be a static method.

<bean id="clientService"

class="examples.ClientService"

factory-method="createInstance"/>

**public** **class** ClientService {

**private** **static** ClientService clientService = **new** ClientService();

**private** ClientService() {}

**public** **static** ClientService createInstance() {

**return** clientService;

}

}

For details about the mechanism for supplying (optional) arguments to the factory method and setting object instance properties after the object is returned from the factory, see [Dependencies and configuration in detail](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-properties-detailed).

#### Instantiation using an instance factory method

Similar to instantiation through a [static factory method](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-class-static-factory-method), instantiation with an instance factory method invokes a non-static method of an existing bean from the container to create a new bean. To use this mechanism, leave the class attribute empty, and in the factory-bean attribute, specify the name of a bean in the current (or parent/ancestor) container that contains the instance method that is to be invoked to create the object. Set the name of the factory method itself with the factory-method attribute.

*<!-- the factory bean, which contains a method called createInstance() -->*

<bean id="serviceLocator" class="examples.DefaultServiceLocator">

*<!-- inject any dependencies required by this locator bean -->*

</bean>

*<!-- the bean to be created via the factory bean -->*

<bean id="clientService"

factory-bean="serviceLocator"

factory-method="createClientServiceInstance"/>

**public** **class** DefaultServiceLocator {

**private** **static** ClientService clientService = **new** ClientServiceImpl();

**private** DefaultServiceLocator() {}

**public** ClientService createClientServiceInstance() {

**return** clientService;

}

}

One factory class can also hold more than one factory method as shown here:

<bean id="serviceLocator" class="examples.DefaultServiceLocator">

*<!-- inject any dependencies required by this locator bean -->*

</bean>

<bean id="clientService"

factory-bean="serviceLocator"

factory-method="createClientServiceInstance"/>

<bean id="accountService"

factory-bean="serviceLocator"

factory-method="createAccountServiceInstance"/>

**public** **class** DefaultServiceLocator {

**private** **static** ClientService clientService = **new** ClientServiceImpl();

**private** **static** AccountService accountService = **new** AccountServiceImpl();

**private** DefaultServiceLocator() {}

**public** ClientService createClientServiceInstance() {

**return** clientService;

}

**public** AccountService createAccountServiceInstance() {

**return** accountService;

}

}

This approach shows that the factory bean itself can be managed and configured through dependency injection (DI). See [Dependencies and configuration in detail.](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-properties-detailed)

|  |
| --- |
| [Note] |
| In Spring documentation, factory bean refers to a bean that is configured in the Spring container that will create objects through an [instance](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-class-instance-factory-method) or [static](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-class-static-factory-method) factory method. By contrast, FactoryBean (notice the capitalization) refers to a Spring-specific [FactoryBean](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-extension-factorybean). |

## 5.4 Dependencies

A typical enterprise application does not consist of a single object (or bean in the Spring parlance). Even the simplest application has a few objects that work together to present what the end-user sees as a coherent application. This next section explains how you go from defining a number of bean definitions that stand alone to a fully realized application where objects collaborate to achieve a goal.

### 5.4.1 Dependency injection

**Dependency injection** (DI) is a process whereby objects define their dependencies, that is, **the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object** instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse(**bean** is deciding for which class he wiil instansiate unlike java code where class were creating the object), hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies on its own by using direct construction of classes, or the Service Locator pattern.

Code is cleaner with the DI principle and decoupling is more effective when **objects are provided with their dependencies**. The object does not look up its dependencies, and does not know the location or class of the dependencies. As such, your classes become easier to test, in particular when the dependencies are on interfaces or abstract base classes, which allow for stub or mock implementations to be used in unit tests.

DI exists in two major variants, [Constructor-based dependency injection](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-constructor-injection) and [Setter-based dependency injection](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-setter-injection).

#### Constructor-based dependency injection

Constructor-based DI is accomplished by the container invoking a constructor with a number of arguments, each representing a **dependency**. Calling a static factory method with specific arguments to construct the bean is nearly equivalent, and this discussion treats arguments to a constructor and to a static factory method similarly. The following example shows a class that can only be dependency-injected with constructor injection. Notice that there is nothing special about this class, it is a POJO that has no dependencies on container specific interfaces, base classes or annotations.

**public** **class** SimpleMovieLister {

*// the SimpleMovieLister has a dependency on a MovieFinder*

**private** MovieFinder movieFinder;

*// a constructor so that the Spring container can 'inject' a MovieFinder*

**public** SimpleMovieLister(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

*// business logic that actually 'uses' the injected MovieFinder is omitted...*

}

##### Constructor argument resolution

Constructor argument resolution matching occurs using the argument's type. If no potential ambiguity exists in the constructor arguments of a bean definition, then the order in which the constructor arguments are defined in a bean definition is the order in which those arguments are supplied to the appropriate constructor when the bean is being instantiated. Consider the following class:

**package** x.y;

**public** **class** Foo {

**public** Foo(Bar bar, Baz baz) {

*// ...*

}

}

No potential ambiguity exists, assuming that Bar and Baz classes are not related by inheritance. Thus the following configuration works fine, and you do not need to specify the constructor argument indexes and/or types explicitly in the <constructor-arg/> element.

<beans>

<bean id="foo" class="x.y.Foo">

<constructor-arg ref="bar"/>

<constructor-arg ref="baz"/>

</bean>

<bean id="bar" class="x.y.Bar"/>

<bean id="baz" class="x.y.Baz"/>

</beans>

When another bean is referenced, the type is known, and matching can occur (as was the case with the preceding example). When a simple type is used, such as <value>true<value>, Spring cannot determine the type of the value, and so cannot match by type without help. Consider the following class:

**package** examples;

**public** **class** ExampleBean {

*// No. of years to the calculate the Ultimate Answer*

**private** **int** years;

*// The Answer to Life, the Universe, and Everything*

**private** String ultimateAnswer;

**public** ExampleBean(**int** years, String ultimateAnswer) {

**this**.years = years;

**this**.ultimateAnswer = ultimateAnswer;

}

}

###### Constructor argument type matching

In the preceding scenario, the container can use type matching with simple types if you explicitly specify the type of the constructor argument using the type attribute. For example:

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg type="int" value="7500000"/>

<constructor-arg type="java.lang.String" value="42"/>

</bean>

###### Constructor argument index

Use the index attribute to specify explicitly the index of constructor arguments. For example:

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg index="0" value="7500000"/>

<constructor-arg index="1" value="42"/>

</bean>

In addition to resolving the ambiguity of multiple simple values, specifying an index resolves ambiguity where a constructor has two arguments of the same type. Note that the index is 0 based.

###### Constructor argument name

As of Spring 3.0 you can also use the constructor parameter name for value disambiguation:

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg name="years" value="7500000"/>

<constructor-arg name="ultimateanswer" value="42"/>

</bean>

Keep in mind that to make this work out of the box your code must be compiled with the debug flag enabled so that Spring can look up the parameter name from the constructor. If you can't compile your code with debug flag (or don't want to) you can use [@ConstructorProperties](https://download.oracle.com/javase/6/docs/api/java/beans/ConstructorProperties.html) JDK annotation to explicitly name your constructor arguments. The sample class would then have to look as follows:

**package** examples;

**public** **class** ExampleBean {

*//* Fields omitted

*@ConstructorProperties({"years", "ultimateAnswer"})*

**public** ExampleBean(**int** years, String ultimateAnswer) {

**this**.years = years;

**this**.ultimateAnswer = ultimateAnswer;

}

}

#### Setter-based dependency injection

Setter-based DI is accomplished by the **container calling setter methods** on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

The following example shows a class that **can only be dependency-injected using pure setter injection**. This class is conventional Java. It is **a POJO that has no dependencies on container specific interfaces, base classes or annotations.**

**public** **class** SimpleMovieLister {

*// the SimpleMovieLister has a dependency on the MovieFinder*

**private** MovieFinder movieFinder;

*// a setter method so that the Spring container can 'inject' a MovieFinder*

**public** **void** setMovieFinder(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

*// business logic that actually 'uses' the injected MovieFinder is omitted...*

}

The ApplicationContext supports constructor- and setter-based DI for the beans it manages. It also supports setter-based DI after some dependencies are already injected through the constructor approach. You configure the dependencies in the form of a BeanDefinition, which you use with PropertyEditor instances to convert properties from one format to another. However, most Spring users do not work with these classes directly (programmatically), but rather with an XML definition file that is then converted internally into instances of these classes, and used to load an entire Spring IoC container instance.

**Constructor-based or setter-based DI?**

Since you can mix both, Constructor- and Setter-based DI, it is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies. Note that the use of a [@Required](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-required-annotation) annotation on a setter can be used to make setters required dependencies.

The **Spring team generally advocates setter injection, because large numbers of constructor arguments can get unwieldy**, especially when properties are optional. **Setter methods also make objects of that class amenable to reconfiguration or re-injection later**. Management through [JMX MBeans](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/jmx.html) is a compelling use case.

Some purists favor constructor-based injection. Supplying all object dependencies means that the object is always returned to client (calling) code in a totally initialized state. The disadvantage is that the object becomes less amenable to reconfiguration and re-injection.

Use the DI that makes the most sense for a particular class. Sometimes, when dealing with third-party classes to which you do not have the source, the choice is made for you. A legacy class may not expose any setter methods, and so constructor injection is the only available DI.

#### Dependency resolution process

The container performs bean dependency resolution as follows:

1. The ApplicationContext is created and initialized with configuration metadata that describes all the beans. Configuration metadata can be specified via XML, Java code or annotations.
2. **For each bean, its dependencies are expressed in the form of properties, constructor arguments, or arguments to the static-factory method if you are using that instead of a normal constructor. These dependencies are provided to the bean, when the bean is actually created**.
3. Each property or constructor argument is an actual definition of the value to set, or a reference to another bean in the container.
4. Each property or constructor argument which is a value is converted from its specified format to the actual type of that property or constructor argument. By default Spring can convert a value supplied in string format to all built-in types, such as int, long, String, boolean, etc.

The Spring container validates the configuration of each bean as the container is created, including the validation of whether bean reference properties refer to valid beans. However, the bean properties themselves are not set until the bean is actually created. Beans that are singleton-scoped and set to be pre-instantiated (the default) are created when the container is created. Scopes are defined in [Section 5.5, “Bean scopes”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes) Otherwise, the bean is created only when it is requested. Creation of a bean potentially causes a graph of beans to be created, as the bean's dependencies and its dependencies' dependencies (and so on) are created and assigned.

**Circular dependencies**

If you use predominantly constructor injection, it is possible to create an unresolvable circular dependency scenario.

For example: Class A requires an instance of class B through constructor injection, and class B requires an instance of class A through constructor injection. If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws a BeanCurrentlyInCreationException.

One possible solution is to edit the source code of some classes to be configured by setters rather than constructors. Alternatively, avoid constructor injection and use setter injection only. In other words, although it is not recommended, you can configure circular dependencies with setter injection.

Unlike the typical case (with no circular dependencies), a circular dependency between bean A and bean B forces one of the beans to be injected into the other prior to being fully initialized itself (a classic chicken/egg scenario).

You can generally trust Spring to do the right thing. It detects configuration problems, such as references to non-existent beans and circular dependencies, at container load-time. Spring sets properties and resolves dependencies as late as possible, when the bean is actually created. This means that a Spring container which has loaded correctly can later generate an exception when you request an object if there is a problem creating that object or one of its dependencies. For example, the bean throws an exception as a result of a missing or invalid property. This potentially delayed visibility of some configuration issues is why ApplicationContext implementations by default pre-instantiate singleton beans. At the cost of some upfront time and memory to create these beans before they are actually needed, you discover configuration issues when the ApplicationContext is created, not later. You can still override this default behavior so that singleton beans will lazy-initialize, rather than be pre-instantiated.

If no circular dependencies exist, when one or more collaborating beans are being injected into a dependent bean, each collaborating bean is totally configured prior to being injected into the dependent bean. This means that if bean A has a dependency on bean B, the Spring IoC container completely configures bean B prior to invoking the setter method on bean A. In other words, the bean is instantiated (if not a pre-instantiated singleton), its dependencies are set, and the relevant lifecycle methods (such as a [configured init method](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-lifecycle-initializingbean) or the [InitializingBean callback method](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-lifecycle-initializingbean)) are invoked.

#### Examples of dependency injection

The following example uses XML-based configuration metadata for setter-based DI. A small part of a Spring XML configuration file specifies some bean definitions:

<bean id="exampleBean" class="examples.ExampleBean">

*<!-- setter injection using the nested <ref/> element -->*

<property name="beanOne"><ref bean="anotherExampleBean"/></property>

*<!-- setter injection using the neater 'ref' attribute -->*

<property name="beanTwo" ref="yetAnotherBean"/>

<property name="integerProperty" value="1"/>

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

**public** **class** ExampleBean {

**private** AnotherBean beanOne;

**private** YetAnotherBean beanTwo;

**private** **int** i;

**public** **void** setBeanOne(AnotherBean beanOne) {

**this**.beanOne = beanOne;

}

**public** **void** setBeanTwo(YetAnotherBean beanTwo) {

**this**.beanTwo = beanTwo;

}

**public** **void** setIntegerProperty(**int** i) {

**this**.i = i;

}

}

In the preceding example, setters are declared to match against the properties specified in the XML file. The following example uses constructor-based DI:

<bean id="exampleBean" class="examples.ExampleBean">

*<!-- constructor injection using the nested <ref/> element -->*

<constructor-arg>

<ref bean="anotherExampleBean"/>

</constructor-arg>

*<!-- constructor injection using the neater 'ref' attribute -->*

<constructor-arg ref="yetAnotherBean"/>

<constructor-arg type="int" value="1"/>

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

**public** **class** ExampleBean {

**private** AnotherBean beanOne;

**private** YetAnotherBean beanTwo;

**private** **int** i;

**public** ExampleBean(

AnotherBean anotherBean, YetAnotherBean yetAnotherBean, **int** i) {

**this**.beanOne = anotherBean;

**this**.beanTwo = yetAnotherBean;

**this**.i = i;

}

}

The constructor arguments specified in the bean definition will be used as arguments to the constructor of the ExampleBean.

Now consider a variant of this example, where instead of using a constructor, Spring is told to call a static factory method to return an instance of the object:

<bean id="exampleBean" class="examples.ExampleBean"

factory-method="createInstance">

<constructor-arg ref="anotherExampleBean"/>

<constructor-arg ref="yetAnotherBean"/>

<constructor-arg value="1"/>

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

**public** **class** ExampleBean {

*// a private constructor*

**private** ExampleBean(...) {

...

}

*// a static factory method; the arguments to this method can be*

*// considered the dependencies of the bean that is returned,*

*// regardless of how those arguments are actually used.*

**public** **static** ExampleBean createInstance (

AnotherBean anotherBean, YetAnotherBean yetAnotherBean, **int** i) {

ExampleBean eb = **new** ExampleBean (...);

*// some other operations...*

**return** eb;

}

}

Arguments to the static factory method are supplied via <constructor-arg/> elements, exactly the same as if a constructor had actually been used. The type of the class being returned by the factory method does not have to be of the same type as the class that contains the static factory method, although in this example it is. An instance (non-static) factory method would be used in an essentially identical fashion (aside from the use of the factory-bean attribute instead of the class attribute), so details will not be discussed here.

### 5.4.2 Dependencies and configuration in detail

As mentioned in the previous section, you can define bean properties and constructor arguments as references to other managed beans (collaborators), or as values defined inline. Spring's XML-based configuration metadata supports sub-element types within its <property/> and <constructor-arg/> elements for this purpose.

#### Straight values (primitives, Strings, and so on)

The value attribute of the <property/> element specifies a property or constructor argument as a human-readable string representation. [As mentioned previously](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/validation.html#beans-beans-conversion), JavaBeans PropertyEditors are used to convert these string values from a String to the actual type of the property or argument.

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

*<!-- results in a setDriverClassName(String) call -->*

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql://localhost:3306/mydb"/>

<property name="username" value="root"/>

<property name="password" value="masterkaoli"/>

</bean>

The following example uses the [p-namespace](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-p-namespace) for even more succinct XML configuration.

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close"

p:driverClassName="com.mysql.jdbc.Driver"

p:url="jdbc:mysql://localhost:3306/mydb"

p:username="root"

p:password="masterkaoli"/>

</beans>

The preceding XML is more succinct; however, typos are discovered at runtime rather than design time, unless you use an IDE such as [IntelliJ IDEA](http://www.jetbrains.com/idea/) or the [SpringSource Tool Suite](http://www.springsource.com/products/sts) (STS) that support automatic property completion when you create bean definitions. Such IDE assistance is highly recommended.

You can also configure a java.util.Properties instance as:

<bean id="mappings"

class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

*<!-- typed as a java.util.Properties -->*

<property name="properties">

<value>

jdbc.driver.className=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/mydb

</value>

</property>

</bean>

The Spring container converts the text inside the <value/> element into a java.util.Properties instance by using the JavaBeans PropertyEditor mechanism. This is a nice shortcut, and is one of a few places where the Spring team do favor the use of the nested <value/> element over the value attribute style.

##### The idref element

The idref element is simply an error-proof way to pass the id (string value - not a reference) of another bean in the container to a <constructor-arg/> or <property/> element.

<bean id="theTargetBean" class="..."/>

<bean id="theClientBean" class="...">

<property name="targetName">

<idref bean="theTargetBean" />

</property>

</bean>

The above bean definition snippet is exactly equivalent (at runtime) to the following snippet:

<bean id="theTargetBean" class="..." />

<bean id="client" class="...">

<property name="targetName" value="theTargetBean" />

</bean>

The first form is preferable to the second, because using the idref tag allows the container to validate at deployment time that the referenced, named bean actually exists. In the second variation, no validation is performed on the value that is passed to the targetName property of the client bean. Typos are only discovered (with most likely fatal results) when the client bean is actually instantiated. If the client bean is a [prototype](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes) bean, this typo and the resulting exception may only be discovered long after the container is deployed.

Additionally, if the referenced bean is in the same XML unit, and the bean name is the bean id, you can use the local attribute, which allows the XML parser itself to validate the bean id earlier, at XML document parse time.

<property name="targetName">

*<!-- a bean with id 'theTargetBean' must exist; otherwise an exception will be thrown -->*

<idref local="theTargetBean"/>

</property>

A common place (at least in versions earlier than Spring 2.0) where the <idref/> element brings value is in the configuration of [AOP interceptors](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/aop-api.html#aop-pfb-1) in a ProxyFactoryBean bean definition. Using <idref/> elements when you specify the interceptor names prevents you from misspelling an interceptor id.

#### References to other beans (collaborators)

The ref element is the final element inside a <constructor-arg/> or <property/> definition element. Here you set the value of the specified property of a bean to be a reference to another bean (a collaborator) managed by the container. The referenced bean is a dependency of the bean whose property will be set, and it is initialized on demand as needed before the property is set. (If the collaborator is a singleton bean, it may be initialized already by the container.) All references are ultimately a reference to another object. Scoping and validation depend on whether you specify the id/name of the other object through the bean,local, or parent attributes.

Specifying the target bean through the bean attribute of the <ref/> tag is the most general form, and allows creation of a reference to any bean in the same container or parent container, regardless of whether it is in the same XML file. The value of the bean attribute may be the same as the id attribute of the target bean, or as one of the values in the name attribute of the target bean.

<ref bean="someBean"/>

Specifying the target bean through the local attribute leverages the ability of the XML parser to validate XML id references within the same file. The value of the local attribute must be the same as the id attribute of the target bean. The XML parser issues an error if no matching element is found in the same file. As such, using the local variant is the best choice (in order to know about errors as early as possible) if the target bean is in the same XML file.

<ref local="someBean"/>

Specifying the target bean through the parent attribute creates a reference to a bean that is in a parent container of the current container. The value of the parent attribute may be the same as either the id attribute of the target bean, or one of the values in the name attribute of the target bean, and the target bean must be in a parent container of the current one. You use this bean reference variant mainly when you have a hierarchy of containers and you want to wrap an existing bean in a parent container with a proxy that will have the same name as the parent bean.

*<!-- in the parent context -->*

<bean id="accountService" class="com.foo.SimpleAccountService">

*<!-- insert dependencies as required as here -->*

</bean>

*<!-- in the child (descendant) context -->*

<bean id="accountService" *<-- bean name is the same as the parent bean -->*

class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="target">

<ref parent="accountService"/> *<!-- notice how we refer to the parent bean -->*

</property>

*<!-- insert other configuration and dependencies as required here -->*

</bean>

#### Inner beans

A <bean/> element inside the <property/> or <constructor-arg/> elements defines a so-called inner bean.

<bean id="outer" class="...">

*<!-- instead of using a reference to a target bean, simply define the target bean inline -->*

<property name="target">

<bean class="com.example.Person"> *<!-- this is the inner bean -->*

<property name="name" value="Fiona Apple"/>

<property name="age" value="25"/>

</bean>

</property>

</bean>

An inner bean definition does not require a defined id or name; the container ignores these values. It also ignores the scope flag. Inner beans are always anonymous and they are always created with the outer bean. It is not possible to inject inner beans into collaborating beans other than into the enclosing bean.

#### Collections

In the <list/>, <set/>, <map/>, and <props/> elements, you set the properties and arguments of the Java Collection types List, Set, Map, and Properties, respectively.

<bean id="moreComplexObject" class="example.ComplexObject">

*<!-- results in a setAdminEmails(java.util.Properties) call -->*

<property name="adminEmails">

<props>

<prop key="administrator">administrator@example.org</prop>

<prop key="support">support@example.org</prop>

<prop key="development">development@example.org</prop>

</props>

</property>

*<!-- results in a setSomeList(java.util.List) call -->*

<property name="someList">

<list>

<value>a list element followed by a reference</value>

<ref bean="myDataSource" />

</list>

</property>

*<!-- results in a setSomeMap(java.util.Map) call -->*

<property name="someMap">

<map>

<entry key="an entry" value="just some string"/>

<entry key ="a ref" value-ref="myDataSource"/>

</map>

</property>

*<!-- results in a setSomeSet(java.util.Set) call -->*

<property name="someSet">

<set>

<value>just some string</value>

<ref bean="myDataSource" />

</set>

</property>

</bean>

The value of a map key or value, or a set value, can also again be any of the following elements:

bean | ref | idref | list | set | map | props | value | null

##### Collection merging

As of Spring 2.0, the container supports the merging of collections. An application developer can define a parent-style <list/>, <map/>, <set/> or <props/> element, and have child-style <list/>, <map/>, <set/> or <props/> elements inherit and override values from the parent collection. That is, the child collection's values are the result of merging the elements of the parent and child collections, with the child's collection elements overriding values specified in the parent collection.

This section on merging discusses the parent-child bean mechanism. Readers unfamiliar with parent and child bean definitions may wish to read the [*relevant section*](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-child-bean-definitions) before continuing.

The following example demonstrates collection merging:

<beans>

<bean id="parent" abstract="true" class="example.ComplexObject">

<property name="adminEmails">

<props>

<prop key="administrator">administrator@example.com</prop>

<prop key="support">support@example.com</prop>

</props>

</property>

</bean>

<bean id="child" parent="parent">

<property name="adminEmails">

*<!-- the merge is specified on the \*child\* collection definition -->*

<props merge="true">

<prop key="sales">sales@example.com</prop>

<prop key="support">support@example.co.uk</prop>

</props>

</property>

</bean>

<beans>

Notice the use of the merge=true attribute on the <props/> element of the adminEmails property of the child bean definition. When the child bean is resolved and instantiated by the container, the resulting instance has an adminEmails Properties collection that contains the result of the merging of the child's adminEmails collection with the parent's adminEmails collection.

administrator=administrator@example.com

sales=sales@example.com

support=support@example.co.uk

The child Properties collection's value set inherits all property elements from the parent <props/>, and the child's value for the support value overrides the value in the parent collection.

This merging behavior applies similarly to the <list/>, <map/>, and <set/> collection types. In the specific case of the <list/> element, the semantics associated with the List collection type, that is, the notion of an ordered collection of values, is maintained; the parent's values precede all of the child list's values. In the case of the Map, Set, and Properties collection types, no ordering exists. Hence no ordering semantics are in effect for the collection types that underlie the associated Map, Set, and Properties implementation types that the container uses internally.

##### Limitations of collection merging

You cannot merge different collection types (such as a Map and a List), and if you do attempt to do so an appropriate Exception is thrown. The merge attribute must be specified on the lower, inherited, child definition; specifying the merge attribute on a parent collection definition is redundant and will not result in the desired merging. The merging feature is available only in Spring 2.0 and later.

##### Strongly-typed collection (Java 5+ only)

In Java 5 and later, you can use strongly typed collections (using generic types). That is, it is possible to declare a Collection type such that it can only contain String elements (for example). If you are using Spring to dependency-inject a strongly-typed Collection into a bean, you can take advantage of Spring's type-conversion support such that the elements of your strongly-typed Collection instances are converted to the appropriate type prior to being added to the Collection.

**public** **class** Foo {

**private** Map<String, Float> accounts;

**public** **void** setAccounts(Map<String, Float> accounts) {

**this**.accounts = accounts;

}

}

<beans>

<bean id="foo" class="x.y.Foo">

<property name="accounts">

<map>

<entry key="one" value="9.99"/>

<entry key="two" value="2.75"/>

<entry key="six" value="3.99"/>

</map>

</property>

</bean>

</beans>

When the accounts property of the foo bean is prepared for injection, the generics information about the element type of the strongly-typed Map<String, Float> is available by reflection. Thus Spring's type conversion infrastructure recognizes the various value elements as being of type Float, and the string values 9.99, 2.75, and 3.99 are converted into an actual Float type.

#### Null and empty string values

Spring treats empty arguments for properties and the like as empty Strings. The following XML-based configuration metadata snippet sets the email property to the empty String value ("")

<bean class="ExampleBean">

<property name="email" value=""/>

</bean>

The preceding example is equivalent to the following Java code: exampleBean.setEmail(""). The <null/> element handles null values. For example:

<bean class="ExampleBean">

<property name="email"><null/></property>

</bean>

The above configuration is equivalent to the following Java code: exampleBean.setEmail(null).

#### XML shortcut with the p-namespace

The p-namespace enables you to use the bean element's attributes, instead of nested <property/> elements, to describe your property values and/or collaborating beans.

Spring 2.0 and later supports extensible configuration formats [with namespaces](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/xsd-config.html), which are based on an XML Schema definition. The beans configuration format discussed in this chapter is defined in an XML Schema document. However, the p-namespace is not defined in an XSD file and exists only in the core of Spring.

The following example shows two XML snippets that resolve to the same result: The first uses standard XML format and the second uses the p-namespace.

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean name="classic" class="com.example.ExampleBean">

<property name="email" value="foo@bar.com"/>

</bean>

<bean name="p-namespace" class="com.example.ExampleBean"

p:email="foo@bar.com"/>

</beans>

The example shows an attribute in the p-namespace called email in the bean definition. This tells Spring to include a property declaration. As previously mentioned, the p-namespace does not have a schema definition, so you can set the name of the attribute to the property name.

This next example includes two more bean definitions that both have a reference to another bean:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean name="john-classic" class="com.example.Person">

<property name="name" value="John Doe"/>

<property name="spouse" ref="jane"/>

</bean>

<bean name="john-modern"

class="com.example.Person"

p:name="John Doe"

p:spouse-ref="jane"/>

<bean name="jane" class="com.example.Person">

<property name="name" value="Jane Doe"/>

</bean>

</beans>

As you can see, this example includes not only a property value using the p-namespace, but also uses a special format to declare property references. Whereas the first bean definition uses <property name="spouse" ref="jane"/> to create a reference from bean john to bean jane, the second bean definition uses p:spouse-ref="jane" as an attribute to do the exact same thing. In this case spouse is the property name, whereas the -ref part indicates that this is not a straight value but rather a reference to another bean.

|  |
| --- |
| [Note] |
| The p-namespace is not as flexible as the standard XML format. For example, the format for declaring property references clashes with properties that end in Ref, whereas the standard XML format does not. We recommend that you choose your approach carefully and communicate this to your team members, to avoid producing XML documents that use all three approaches at the same time. |

#### XML shortcut with the c-namespace

Similar to the [the section called “XML shortcut with the p-namespace”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-p-namespace), the c-namespace, newly introduced in Spring 3.1, allows usage of inlined attributes for configuring the constructor arguments rather then nested constructor-arg elements.

Let's review the examples from [the section called “Constructor-based dependency injection”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-constructor-injection) with the c namespace:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:c="http://www.springframework.org/schema/c"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http:*//www.springframework.org/schema/beans/spring-beans.xsd">*

<bean id="bar" **class**="x.y.Bar"/>

<bean id="baz" **class**="x.y.Baz"/>

<-- 'traditional' declaration -->

<bean id="foo" **class**="x.y.Foo">

<constructor-arg ref="bar"/>

<constructor-arg ref="baz"/>

<constructor-arg value="foo@bar.com"/>

</bean>

<-- 'c-namespace' declaration -->

<bean id="foo" **class**="x.y.Foo" c:bar-ref="bar" c:baz-ref="baz" c:email="foo@bar.com">

</beans>

The c: namespace uses the same conventions as the p: one (trailing -ref for bean references) for setting the constructor arguments by their names. And just as well, it needs to be declared even though it is not defined in an XSD schema (but it exists inside the Spring core).

For the rare cases where the constructor argument names are not available (usually if the bytecode was compiled without debugging information), one can use fallback to the argument indexes:

<-- 'c-namespace' index declaration -->

<bean id="foo" **class**="x.y.Foo" c:\_0-ref="bar" c:\_1-ref="baz">

|  |
| --- |
| [Note] |
| Due to the XML grammar, the index notation requires the presence of the leading \_ as XML attribute names cannot start with a number (even though some IDE allow it). |

In practice, the constructor resolution [mechanism](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-ctor-arguments-resolution) is quite efficient in matching arguments so unless one really needs to, we recommend using the name notation through-out your configuration.

#### Compound property names

You can use compound or nested property names when you set bean properties, as long as all components of the path except the final property name are not null. Consider the following bean definition.

<bean id="foo" class="foo.Bar">

<property name="fred.bob.sammy" value="123" />

</bean>

The foo bean has a fred property, which has a bob property, which has a sammy property, and that final sammy property is being set to the value 123. In order for this to work, the fred property of foo, and the bob property of fred must not be null after the bean is constructed, or a NullPointerException is thrown.

### 5.4.3 Using depends-on

If a bean is a dependency of another that usually means that one bean is set as a property of another. Typically you accomplish this with the [<ref/> element](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-ref-element) in XML-based configuration metadata. However, sometimes dependencies between beans are less direct; for example, a static initializer in a class needs to be triggered, such as database driver registration. The depends-on attribute can explicitly force one or more beans to be initialized before the bean using this element is initialized. The following example uses the depends-on attribute to express a dependency on a single bean:

<bean id="beanOne" class="ExampleBean" depends-on="**manager**"/>

<bean id="**manager**" class="ManagerBean" />

To express a dependency on multiple beans, supply a list of bean names as the value of the depends-on attribute, with commas, whitespace and semicolons, used as valid delimiters:

<bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">

<property name="manager" ref="manager" />

</bean>

<bean id="manager" class="ManagerBean" />

<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />

|  |
| --- |
| [Note] |
| The depends-on attribute in the bean definition can specify both an initialization time dependency and, in the case of [singleton](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes-singleton) beans only, a corresponding destroy time dependency. Dependent beans that define a depends-on relationship with a given bean are destroyed first, prior to the given bean itself being destroyed. Thus depends-on can also control shutdown order. |

### 5.4.4 Lazy-initialized beans

By default, ApplicationContext implementations eagerly create and configure all [singleton](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes-singleton) beans as part of the initialization process. Generally, this pre-instantiation is desirable, because errors in the configuration or surrounding environment are discovered immediately, as opposed to hours or even days later. When this behavior is not desirable, you can prevent pre-instantiation of a singleton bean by marking the bean definition as lazy-initialized. A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

In XML, this behavior is controlled by the lazy-init attribute on the <bean/> element; for example:

<bean id="lazy" class="com.foo.ExpensiveToCreateBean" **lazy-init="true"**/>

<bean name="not.lazy" class="com.foo.AnotherBean"/>

When the preceding configuration is consumed by an ApplicationContext, the bean named lazy is not eagerly pre-instantiated when the ApplicationContext is starting up, whereas the not.lazy bean is eagerly pre-instantiated.

However, when a lazy-initialized bean is a dependency of a singleton bean that is not lazy-initialized, the ApplicationContext creates the lazy-initialized bean at startup, because it must satisfy the singleton's dependencies. The lazy-initialized bean is injected into a singleton bean elsewhere that is not lazy-initialized.

You can also control lazy-initialization at the container level by using the default-lazy-init attribute on the <beans/> element; for example:

<beans default-lazy-init="true">

*<!-- no beans will be pre-instantiated... -->*

</beans>

### 5.4.5 Autowiring collaborators

The Spring container can autowire relationships between collaborating beans. You can allow Spring to resolve collaborators (other beans) automatically for your bean by inspecting the contents of the ApplicationContext. Autowiring has the following advantages:

* Autowiring can significantly reduce the need to specify properties or constructor arguments. (Other mechanisms such as a bean template [discussed elsewhere in this chapter](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-child-bean-definitions) are also valuable in this regard.)
* Autowiring can update a configuration as your objects evolve. For example, if you need to add a dependency to a class, that dependency can be satisfied automatically without you needing to modify the configuration. Thus autowiring can be especially useful during development, without negating the option of switching to explicit wiring when the code base becomes more stable.

When using XML-based configuration metadata[[2]](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html" \l "ftn.d5e1977), you specify autowire mode for a bean definition with the autowire attribute of the <bean/> element. The autowiring functionality has five modes. You specify autowiring per bean and thus can choose which ones to autowire.

**Table 5.2. Autowiring modes**

| **Mode** | **Explanation** |
| --- | --- |
| no | (Default) No autowiring. Bean references must be defined via a ref element. Changing the default setting is not recommended for larger deployments, because specifying collaborators explicitly gives greater control and clarity. To some extent, it documents the structure of a system. |
| byName | Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired. For example, if a bean definition is set to autowire by name, and it contains a master property (that is, it has a setMaster(..) method), Spring looks for a bean definition named master, and uses it to set the property. |
| byType | Allows a property to be autowired if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown, which indicates that you may not use byType autowiring for that bean. If there are no matching beans, nothing happens; the property is not set. |
| constructor | Analogous to byType, but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised. |

With byType or constructor autowiring mode, you can wire arrays and typed-collections. In such cases all autowire candidates within the container that match the expected type are provided to satisfy the dependency. You can autowire strongly-typed Maps if the expected key type is String. An autowired Maps values will consist of all bean instances that match the expected type, and the Maps keys will contain the corresponding bean names.

You can combine autowire behavior with dependency checking, which is performed after autowiring completes.

#### Limitations and disadvantages of autowiring

Autowiring works best when it is used consistently across a project. If autowiring is not used in general, it might be confusing to developers to use it to wire only one or two bean definitions.

Consider the limitations and disadvantages of autowiring:

* Explicit dependencies in property and constructor-arg settings always override autowiring. You cannot autowire so-called simple properties such as primitives, Strings, and Classes (and arrays of such simple properties). This limitation is by-design.
* Autowiring is less exact than explicit wiring. Although, as noted in the above table, Spring is careful to avoid guessing in case of ambiguity that might have unexpected results, the relationships between your Spring-managed objects are no longer documented explicitly.
* Wiring information may not be available to tools that may generate documentation from a Spring container.
* Multiple bean definitions within the container may match the type specified by the setter method or constructor argument to be autowired. For arrays, collections, or Maps, this is not necessarily a problem. However for dependencies that expect a single value, this ambiguity is not arbitrarily resolved. If no unique bean definition is available, an exception is thrown.

In the latter scenario, you have several options:

* Abandon autowiring in favor of explicit wiring.
* Avoid autowiring for a bean definition by setting its autowire-candidate attributes to false as described in the next section.
* Designate a single bean definition as the primary candidate by setting the primary attribute of its <bean/> element to true.
* If you are using Java 5 or later, implement the more fine-grained control available with annotation-based configuration, as described in [Section 5.9, “Annotation-based container configuration”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-annotation-config).

#### Excluding a bean from autowiring

On a per-bean basis, you can exclude a bean from autowiring. In Spring's XML format, set the autowire-candidate attribute of the <bean/> element to false; the container makes that specific bean definition unavailable to the autowiring infrastructure (including annotation style configurations such as [@Autowired](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-autowired-annotation)).

You can also limit autowire candidates based on pattern-matching against bean names. The top-level <beans/> element accepts one or more patterns within its default-autowire-candidates attribute. For example, to limit autowire candidate status to any bean whose name ends with Repository, provide a value of \*Repository. To provide multiple patterns, define them in a comma-separated list. An explicit value of true or false for a bean definitions autowire-candidate attribute always takes precedence, and for such beans, the pattern matching rules do not apply.

These techniques are useful for beans that you never want to be injected into other beans by autowiring. It does not mean that an excluded bean cannot itself be configured using autowiring. Rather, the bean itself is not a candidate for autowiring other beans.

### 5.4.6 Method injection

In most application scenarios, most beans in the container are [singletons](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes-singleton). When a singleton bean needs to collaborate with another singleton bean, or a non-singleton bean needs to collaborate with another non-singleton bean, you typically handle the dependency by defining one bean as a property of the other. A problem arises when the bean lifecycles are different. Suppose singleton bean A needs to use non-singleton (prototype) bean B, perhaps on each method invocation on A. The container only creates the singleton bean A once, and thus only gets one opportunity to set the properties. The container cannot provide bean A with a new instance of bean B every time one is needed.

A solution is to forego some inversion of control. You can [make bean A aware of the container](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-aware) by implementing the ApplicationContextAware interface, and by [making a getBean("B") call to the container](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-client) ask for (a typically new) bean B instance every time bean A needs it. The following is an example of this approach:

*// a class that uses a stateful Command-style class to perform some processing*

**package** fiona.apple;

*// Spring-API imports*

**import** org.springframework.beans.BeansException;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.ApplicationContextAware;

**public** **class** CommandManager **implements** ApplicationContextAware {

**private** ApplicationContext applicationContext;

**public** Object process(Map commandState) {

*// grab a new instance of the appropriate Command*

Command command = createCommand();

*// set the state on the (hopefully brand new) Command instance*

command.setState(commandState);

**return** command.execute();

}

**protected** Command createCommand() {

*// notice the Spring API dependency!*

**return** **this**.applicationContext.getBean("command", Command.**class**);

}

**public** **void** setApplicationContext(ApplicationContext applicationContext)

**throws** BeansException {

**this**.applicationContext = applicationContext;

}

}

The preceding is not desirable, because the business code is aware of and coupled to the Spring Framework. Method Injection, a somewhat advanced feature of the Spring IoC container, allows this use case to be handled in a clean fashion.

You can read more about the motivation for Method Injection in [this blog entry](http://blog.springsource.com/2004/08/06/method-injection/).

#### Lookup method injection

Lookup method injection is the ability of the container to override methods on container managed beans, to return the lookup result for another named bean in the container. The lookup typically involves a prototype bean as in the scenario described in the preceding section. The Spring Framework implements this method injection by using bytecode generation from the CGLIB library to generate dynamically a subclass that overrides the method.

|  |
| --- |
| [Note] |
| For this dynamic subclassing to work, the class that the Spring container will subclass cannot be final, and the method to be overridden cannot be final either. Also, testing a class that has an abstract method requires you to subclass the class yourself and to supply a stub implementation of the abstract method. Finally, objects that have been the target of method injection cannot be serialized. As of Spring 3.2 it is no longer necessary to add CGLIB to your classpath, because CGLIB classes are repackaged under org.springframework and distributed within the spring-core JAR. This is done both for convenience as well as to avoid potential conflicts with other projects that use differing versions of CGLIB. |

Looking at the CommandManager class in the previous code snippet, you see that the Spring container will dynamically override the implementation of the createCommand() method. Your CommandManager class will not have any Spring dependencies, as can be seen in the reworked example:

**package** fiona.apple;

*// no more Spring imports!*

**public** **abstract** **class** CommandManager {

**public** Object process(Object commandState) {

*// grab a new instance of the appropriate Command interface*

Command command = createCommand();

*// set the state on the (hopefully brand new) Command instance*

command.setState(commandState);

**return** command.execute();

}

*// okay... but where is the implementation of this method?*

**protected** **abstract** Command createCommand();

}

In the client class containing the method to be injected (the CommandManager in this case), the method to be injected requires a signature of the following form:

<public|protected> [abstract] <return-type> theMethodName(*no-arguments*);

If the method is abstract, the dynamically-generated subclass implements the method. Otherwise, the dynamically-generated subclass overrides the concrete method defined in the original class. For example:

*<!-- a stateful bean deployed as a prototype (non-singleton) -->*

<bean id="command" class="fiona.apple.AsyncCommand" scope="prototype">

*<!-- inject dependencies here as required -->*

</bean>

*<!-- commandProcessor uses statefulCommandHelper -->*

<bean id="commandManager" class="fiona.apple.CommandManager">

<lookup-method name="createCommand" bean="command"/>

</bean>

The bean identified as commandManager calls its own method createCommand() whenever it needs a new instance of the command bean. You must be careful to deploy the command bean as a prototype, if that is actually what is needed. If it is deployed as a [singleton](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes-singleton), the same instance of the command bean is returned each time.

|  |
| --- |
| [Tip] |
| The interested reader may also find the ServiceLocatorFactoryBean (in the org.springframework.beans.factory.config package) to be of use. The approach used in ServiceLocatorFactoryBean is similar to that of another utility class, ObjectFactoryCreatingFactoryBean, but it allows you to specify your own lookup interface as opposed to a Spring-specific lookup interface. Consult the JavaDocs for these classes as well as this [blog entry](http://blog.arendsen.net/index.php/2006/10/05/on-the-servicelocatorfactorybean-dlas-and-the-sustainability-of-code-and-design/) for additional information ServiceLocatorFactoryBean. |

#### Arbitrary method replacement

A less useful form of method injection than lookup method Injection is the ability to replace arbitrary methods in a managed bean with another method implementation. Users may safely skip the rest of this section until the functionality is actually needed.

With XML-based configuration metadata, you can use the replaced-method element to replace an existing method implementation with another, for a deployed bean. Consider the following class, with a method computeValue, which we want to override:

**public** **class** MyValueCalculator {

**public** String computeValue(String input) {

*// some real code...*

}

*// some other methods...*

}

A class implementing the org.springframework.beans.factory.support.MethodReplacer interface provides the new method definition.

*/\*\* meant to be used to override the existing computeValue(String)*

*implementation in MyValueCalculator*

*\*/*

**public** **class** ReplacementComputeValue **implements** MethodReplacer {

**public** Object reimplement(Object o, Method m, Object[] args) **throws** Throwable {

*// get the input value, work with it, and return a computed result*

String input = (String) args[0];

...

**return** ...;

}

}

The bean definition to deploy the original class and specify the method override would look like this:

<bean id="myValueCalculator" class="x.y.z.MyValueCalculator">

*<!-- arbitrary method replacement -->*

<replaced-method name="computeValue" replacer="replacementComputeValue">

<arg-type>String</arg-type>

</replaced-method>

</bean>

<bean id="replacementComputeValue" class="a.b.c.ReplacementComputeValue"/>

You can use one or more contained <arg-type/> elements within the <replaced-method/> element to indicate the method signature of the method being overridden. The signature for the arguments is necessary only if the method is overloaded and multiple variants exist within the class. For convenience, the type string for an argument may be a substring of the fully qualified type name. For example, the following all match java.lang.String:

java.lang.String

String

Str

Because the number of arguments is often enough to distinguish between each possible choice, this shortcut can save a lot of typing, by allowing you to type only the shortest string that will match an argument type.

## 5.5 Bean scopes

When you create a bean definition, you create a recipe for creating actual instances of the class defined by that bean definition. The idea that a bean definition is a recipe is important, because it means that, as with a class, you can create many object instances from a single recipe.

You can control not only the various dependencies and configuration values that are to be plugged into an object that is created from a particular bean definition, but also the scope of the objects created from a particular bean definition. This approach is powerful and flexible in that you can choose the scope of the objects you create through configuration instead of having to bake in the scope of an object at the Java class level. Beans can be defined to be deployed in one of a number of scopes: out of the box, the Spring Framework supports five scopes, three of which are available only if you use a web-aware ApplicationContext.

The following scopes are supported out of the box. You can also create [a custom scope.](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes-custom)

**Table 5.3. Bean scopes**

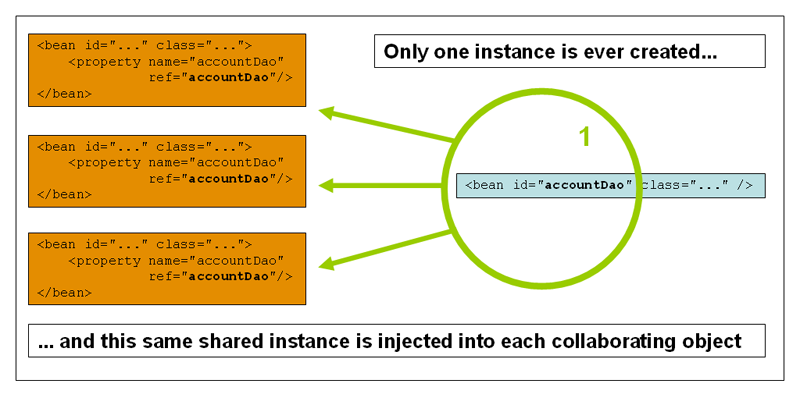
| **Scope** | **Description** |
| --- | --- |
| [singleton](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes-singleton) | (Default) Scopes a single bean definition to a single object instance per Spring IoC container. |
| [prototype](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes-prototype) | Scopes a single bean definition to any number of object instances. |
| [request](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes-request) | Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext. |
| [session](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes-session) | Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext. |
| [global session](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes-global-session) | Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext. |

|  |
| --- |
| [Note] |
| As of Spring 3.0, a thread scope is available, but is not registered by default. For more information, see the documentation for [SimpleThreadScope](http://static.springsource.org/spring/docs/current/api/org/springframework/context/support/SimpleThreadScope.html). For instructions on how to register this or any other custom scope, see [the section called “Using a custom scope”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes-custom-using). |

### 5.5.1 The singleton scope

Only one shared instance of a singleton bean is managed, and all requests for beans with an id or ids matching that bean definition result in that one specific bean instance being returned by the Spring container.

To put it another way, when you define a bean definition and it is scoped as a singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.



Spring's concept of a singleton bean differs from the Singleton pattern as defined in the Gang of Four (GoF) patterns book. The GoF Singleton hard-codes the scope of an object such that one and only one instance of a particular class is created per *ClassLoader*. The scope of the Spring singleton is best described as per container and per bean. This means that if you define one bean for a particular class in a single Spring container, then the Spring container creates one and only one instance of the class defined by that bean definition. The singleton scope is the default scope in Spring. To define a bean as a singleton in XML, you would write, for example:

<bean id="accountService" class="com.foo.DefaultAccountService"/>

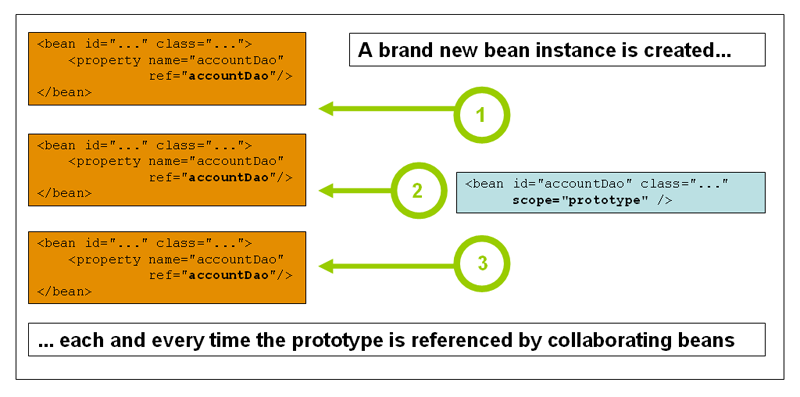
*<!-- the following is equivalent, though redundant (singleton scope is the default) -->*

<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>

### 5.5.2 The prototype scope

The non-singleton, prototype scope of bean deployment results in the creation of a new bean instance every time a request for that specific bean is made. That is, the bean is injected into another bean or you request it through a getBean() method call on the container. As a rule, use the prototype scope for all stateful beans and the singleton scope for stateless beans.

The following diagram illustrates the Spring prototype scope. A data access object (DAO) is not typically configured as a prototype, because a typical DAO does not hold any conversational state; it was just easier for this author to reuse the core of the singleton diagram.



The following example defines a bean as a prototype in XML:

*<!-- using spring-beans-2.0.dtd -->*

<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>

In contrast to the other scopes, Spring does not manage the complete lifecycle of a prototype bean: the container instantiates, configures, and otherwise assembles a prototype object, and hands it to the client, with no further record of that prototype instance. Thus, although initialization lifecycle callback methods are called on all objects regardless of scope, in the case of prototypes, configured destruction lifecycle callbacks are not called. The client code must clean up prototype-scoped objects and release expensive resources that the prototype bean(s) are holding. To get the Spring container to release resources held by prototype-scoped beans, try using a custom [bean post-processor](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-extension-bpp), which holds a reference to beans that need to be cleaned up.

In some respects, the Spring container's role in regard to a prototype-scoped bean is a replacement for the Java new operator. All lifecycle management past that point must be handled by the client. (For details on the lifecycle of a bean in the Spring container, see [Section 5.6.1, “Lifecycle callbacks”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-lifecycle).)

### 5.5.3 Singleton beans with prototype-bean dependencies

When you use singleton-scoped beans with dependencies on prototype beans, be aware that dependencies are resolved at instantiation time. Thus if you dependency-inject a prototype-scoped bean into a singleton-scoped bean, a new prototype bean is instantiated and then dependency-injected into the singleton bean. The prototype instance is the sole instance that is ever supplied to the singleton-scoped bean.

However, suppose you want the singleton-scoped bean to acquire a new instance of the prototype-scoped bean repeatedly at runtime. You cannot dependency-inject a prototype-scoped bean into your singleton bean, because that injection occurs only once, when the Spring container is instantiating the singleton bean and resolving and injecting its dependencies. If you need a new instance of a prototype bean at runtime more than once, see [Section 5.4.6, “Method injection”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-method-injection)

### 5.5.4 Request, session, and global session scopes

The request, session, and global session scopes are only available if you use a web-aware Spring ApplicationContext implementation (such as XmlWebApplicationContext). If you use these scopes with regular Spring IoC containers such as the ClassPathXmlApplicationContext, you get an IllegalStateException complaining about an unknown bean scope.

#### Initial web configuration

To support the scoping of beans at the request, session, and global session levels (web-scoped beans), some minor initial configuration is required before you define your beans. (This initial setup is not required for the standard scopes, singleton and prototype.)

How you accomplish this initial setup depends on your particular Servlet environment..

If you access scoped beans within Spring Web MVC, in effect, within a request that is processed by the Spring DispatcherServlet, or DispatcherPortlet, then no special setup is necessary: DispatcherServlet and DispatcherPortlet already expose all relevant state.

If you use a Servlet 2.4+ web container, with requests processed outside of Spring's DispatcherServlet (for example, when using JSF or Struts), you need to add the following javax.servlet.ServletRequestListener to the declarations in your web applications web.xml file:

<web-app>

...

<listener>

<listener-class>

org.springframework.web.context.request.RequestContextListener

</listener-class>

</listener>

...

</web-app>

If you use an older web container (Servlet 2.3), use the provided javax.servlet.Filter implementation. The following snippet of XML configuration must be included in the web.xml file of your web application if you want to access web-scoped beans in requests outside of Spring's DispatcherServlet on a Servlet 2.3 container. (The filter mapping depends on the surrounding web application configuration, so you must change it as appropriate.)

<web-app>

..

<filter>

<filter-name>requestContextFilter</filter-name>

<filter-class>org.springframework.web.filter.RequestContextFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>requestContextFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

...

</web-app>

DispatcherServlet, RequestContextListener and RequestContextFilter all do exactly the same thing, namely bind the HTTP request object to the Thread that is servicing that request. This makes beans that are request- and session-scoped available further down the call chain.

#### Request scope

Consider the following bean definition:

<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>

The Spring container creates a new instance of the LoginAction bean by using the loginAction bean definition for each and every HTTP request. That is, the loginAction bean is scoped at the HTTP request level. You can change the internal state of the instance that is created as much as you want, because other instances created from the same loginAction bean definition will not see these changes in state; they are particular to an individual request. When the request completes processing, the bean that is scoped to the request is discarded.

#### Session scope

Consider the following bean definition:

<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>

The Spring container creates a new instance of the UserPreferences bean by using the userPreferences bean definition for the lifetime of a single HTTP Session. In other words, the userPreferences bean is effectively scoped at the HTTP Session level. As with request-scoped beans, you can change the internal state of the instance that is created as much as you want, knowing that other HTTP Session instances that are also using instances created from the same userPreferences bean definition do not see these changes in state, because they are particular to an individual HTTP Session. When the HTTP Session is eventually discarded, the bean that is scoped to that particular HTTP Session is also discarded.

#### Global session scope

Consider the following bean definition:

<bean id="userPreferences" class="com.foo.UserPreferences" scope="globalSession"/>

The global session scope is similar to the standard HTTP Session scope ([described above](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes-session)), and applies only in the context of portlet-based web applications. The portlet specification defines the notion of a global Session that is shared among all portlets that make up a single portlet web application. Beans defined at the global session scope are scoped (or bound) to the lifetime of the global portlet Session.

If you write a standard Servlet-based web application and you define one or more beans as having global session scope, the standard HTTP Session scope is used, and no error is raised.

#### Scoped beans as dependencies

The Spring IoC container manages not only the instantiation of your objects (beans), but also the wiring up of collaborators (or dependencies). If you want to inject (for example) an HTTP request scoped bean into another bean, you must inject an AOP proxy in place of the scoped bean. That is, you need to inject a proxy object that exposes the same public interface as the scoped object but that can also retrieve the real, target object from the relevant scope (for example, an HTTP request) and delegate method calls onto the real object.

|  |
| --- |
| [Note] |
| You do not need to use the <aop:scoped-proxy/> in conjunction with beans that are scoped as singletons or prototypes. |

The configuration in the following example is only one line, but it is important to understand the “why” as well as the “how” behind it.

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

*<!-- an HTTP Session-scoped bean exposed as a proxy -->*

<bean id="userPreferences" class="com.foo.UserPreferences" scope="session">

*<!-- instructs the container to proxy the surrounding bean -->*

<aop:scoped-proxy/>

</bean>

*<!-- a singleton-scoped bean injected with a proxy to the above bean -->*

<bean id="userService" class="com.foo.SimpleUserService">

*<!-- a reference to the proxied userPreferences bean -->*

<property name="userPreferences" ref="userPreferences"/>

</bean>

</beans>

To create such a proxy, you insert a child <aop:scoped-proxy/> element into a scoped bean definition. See [the section called “Choosing the type of proxy to create”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes-other-injection-proxies) and [Appendix E, *XML Schema-based configuration*](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/xsd-config.html).) Why do definitions of beans scoped at the request, session, globalSession and custom-scope levels require the <aop:scoped-proxy/> element ? Let's examine the following singleton bean definition and contrast it with what you need to define for the aforementioned scopes. (The following userPreferences bean definition as it stands is incomplete.)

<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>

<bean id="userManager" class="com.foo.UserManager">

<property name="userPreferences" ref="userPreferences"/>

</bean>

In the preceding example, the singleton bean userManager is injected with a reference to the HTTP Session-scoped bean userPreferences. The salient point here is that the userManager bean is a singleton: it will be instantiated exactly once per container, and its dependencies (in this case only one, the userPreferences bean) are also injected only once. This means that the userManager bean will only operate on the exact same userPreferences object, that is, the one that it was originally injected with.

This is not the behavior you want when injecting a shorter-lived scoped bean into a longer-lived scoped bean, for example injecting an HTTP Session-scoped collaborating bean as a dependency into singleton bean. Rather, you need a single userManager object, and for the lifetime of an HTTP Session, you need a userPreferences object that is specific to said HTTP Session. Thus the container creates an object that exposes the exact same public interface as the UserPreferences class (ideally an object that is a UserPreferences instance) which can fetch the real UserPreferences object from the scoping mechanism (HTTP request, Session, etc.). The container injects this proxy object into the userManager bean, which is unaware that this UserPreferences reference is a proxy. In this example, when a UserManager instance invokes a method on the dependency-injected UserPreferences object, it actually is invoking a method on the proxy. The proxy then fetches the real UserPreferences object from (in this case) the HTTP Session, and delegates the method invocation onto the retrieved real UserPreferences object.

Thus you need the following, correct and complete, configuration when injecting request-, session-, and globalSession-scoped beans into collaborating objects:

<bean id="userPreferences" class="com.foo.UserPreferences" scope="session">

<aop:scoped-proxy/>

</bean>

<bean id="userManager" class="com.foo.UserManager">

<property name="userPreferences" ref="userPreferences"/>

</bean>

##### Choosing the type of proxy to create

By default, when the Spring container creates a proxy for a bean that is marked up with the <aop:scoped-proxy/> element, a CGLIB-based class proxy is created.

Note: CGLIB proxies only intercept public method calls! Do not call non-public methods on such a proxy; they will not be delegated to the scoped target object.

Alternatively, you can configure the Spring container to create standard JDK interface-based proxies for such scoped beans, by specifying false for the value of the proxy-target-class attribute of the <aop:scoped-proxy/> element. Using JDK interface-based proxies means that you do not need additional libraries in your application classpath to effect such proxying. However, it also means that the class of the scoped bean must implement at least one interface, and that all collaborators into which the scoped bean is injected must reference the bean through one of its interfaces.

*<!-- DefaultUserPreferences implements the UserPreferences interface -->*

<bean id="userPreferences" class="com.foo.DefaultUserPreferences" scope="session">

<aop:scoped-proxy proxy-target-class="false"/>

</bean>

<bean id="userManager" class="com.foo.UserManager">

<property name="userPreferences" ref="userPreferences"/>

</bean>

For more detailed information about choosing class-based or interface-based proxying, see [Section 9.6, “Proxying mechanisms”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/aop.html#aop-proxying).

### 5.5.5 Custom scopes

As of Spring 2.0, the bean scoping mechanism is extensible. You can define your own scopes, or even redefine existing scopes, although the latter is considered bad practice and you cannot override the built-in singleton and prototype scopes.

#### Creating a custom scope

To integrate your custom scope(s) into the Spring container, you need to implement the org.springframework.beans.factory.config.Scope interface, which is described in this section. For an idea of how to implement your own scopes, see the Scope implementations that are supplied with the Spring Framework itself and the [Scope Javadoc](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/Scope.html), which explains the methods you need to implement in more detail.

The Scope interface has four methods to get objects from the scope, remove them from the scope, and allow them to be destroyed.

The following method returns the object from the underlying scope. The session scope implementation, for example, returns the session-scoped bean (and if it does not exist, the method returns a new instance of the bean, after having bound it to the session for future reference).

Object get(String name, ObjectFactory objectFactory)

The following method removes the object from the underlying scope. The session scope implementation for example, removes the session-scoped bean from the underlying session. The object should be returned, but you can return null if the object with the specified name is not found.

Object remove(String name)

The following method registers the callbacks the scope should execute when it is destroyed or when the specified object in the scope is destroyed. Refer to the Javadoc or a Spring scope implementation for more information on destruction callbacks.

**void** registerDestructionCallback(String name, Runnable destructionCallback)

The following method obtains the conversation identifier for the underlying scope. This identifier is different for each scope. For a session scoped implementation, this identifier can be the session identifier.

String getConversationId()

#### Using a custom scope

After you write and test one or more custom Scope implementations, you need to make the Spring container aware of your new scope(s). The following method is the central method to register a new Scope with the Spring container:

**void** registerScope(String scopeName, Scope scope);

This method is declared on the ConfigurableBeanFactory interface, which is available on most of the concrete ApplicationContext implementations that ship with Spring via the BeanFactory property.

The first argument to the registerScope(..) method is the unique name associated with a scope; examples of such names in the Spring container itself are singleton and prototype. The second argument to the registerScope(..) method is an actual instance of the custom Scope implementation that you wish to register and use.

Suppose that you write your custom Scope implementation, and then register it as below.

|  |
| --- |
| [Note] |
| The example below uses SimpleThreadScope which is included with Spring, but not registered by default. The instructions would be the same for your own custom Scope implementations. |

Scope threadScope = **new** SimpleThreadScope();

beanFactory.registerScope("thread", threadScope);

You then create bean definitions that adhere to the scoping rules of your custom Scope:

<bean id="..." class="..." scope="thread">

With a custom Scope implementation, you are not limited to programmatic registration of the scope. You can also do the Scope registration declaratively, using the CustomScopeConfigurer class:

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">

<property name="scopes">

<map>

<entry key="thread">

<bean class="org.springframework.context.support.SimpleThreadScope"/>

</entry>

</map>

</property>

</bean>

<bean id="bar" class="x.y.Bar" scope="thread">

<property name="name" value="Rick"/>

<aop:scoped-proxy/>

</bean>

<bean id="foo" class="x.y.Foo">

<property name="bar" ref="bar"/>

</bean>

</beans>

|  |
| --- |
| [Note] |
| When you place <aop:scoped-proxy/> in a FactoryBean implementation, it is the factory bean itself that is scoped, not the object returned from getObject(). |

## 5.6 Customizing the nature of a bean

### 5.6.1 Lifecycle callbacks

To interact with the container's management of the bean lifecycle, you can implement the Spring InitializingBean and DisposableBean interfaces. The container calls afterPropertiesSet() for the former and destroy() for the latter to allow the bean to perform certain actions upon initialization and destruction of your beans.

|  |
| --- |
| [Tip] |
| The JSR-250 @PostConstruct and @PreDestroy annotations are generally considered best practice for receiving lifecycle callbacks in a modern Spring application. Using these annotations means that your beans are not coupled to Spring specific interfaces. For details see [Section 5.9.6, “@PostConstruct and @PreDestroy”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-postconstruct-and-predestroy-annotations).  If you don't want to use the JSR-250 annotations but you are still looking to remove coupling consider the use of init-method and destroy-method object definition metadata. |

Internally, the Spring Framework uses BeanPostProcessor implementations to process any callback interfaces it can find and call the appropriate methods. If you need custom features or other lifecycle behavior Spring does not offer out-of-the-box, you can implement a BeanPostProcessor yourself. For more information, see [Section 5.8, “Container Extension Points”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-extension).

In addition to the initialization and destruction callbacks, Spring-managed objects may also implement the Lifecycle interface so that those objects can participate in the startup and shutdown process as driven by the container's own lifecycle.

The lifecycle callback interfaces are described in this section.

#### Initialization callbacks

The org.springframework.beans.factory.InitializingBean interface allows a bean to perform initialization work after all necessary properties on the bean have been set by the container. The InitializingBean interface specifies a single method:

**void** afterPropertiesSet() **throws** Exception;

It is recommended that you do not use the InitializingBean interface because it unnecessarily couples the code to Spring. Alternatively, use the [@PostConstruct annotation](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-postconstruct-and-predestroy-annotations) or specify a POJO initialization method. In the case of XML-based configuration metadata, you use the init-method attribute to specify the name of the method that has a void no-argument signature. For example, the following definition:

<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>

**public** **class** ExampleBean {

**public** **void** init() {

*// do some initialization work*

}

}

...is exactly the same as...

<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>

**public** **class** AnotherExampleBean **implements** InitializingBean {

**public** **void** afterPropertiesSet() {

*// do some initialization work*

}

}

... but does not couple the code to Spring.

#### Destruction callbacks

Implementing the org.springframework.beans.factory.DisposableBean interface allows a bean to get a callback when the container containing it is destroyed. The DisposableBean interface specifies a single method:

**void** destroy() **throws** Exception;

It is recommended that you do not use the DisposableBean callback interface because it unnecessarily couples the code to Spring. Alternatively, use the [@PreDestroy annotation](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-postconstruct-and-predestroy-annotations) or specify a generic method that is supported by bean definitions. With XML-based configuration metadata, you use the destroy-method attribute on the <bean/>. For example, the following definition:

<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>

**public** **class** ExampleBean {

**public** **void** cleanup() {

*// do some destruction work (like releasing pooled connections)*

}

}

...is exactly the same as...

<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>

**public** **class** AnotherExampleBean **implements** DisposableBean {

**public** **void** destroy() {

*// do some destruction work (like releasing pooled connections)*

}

}

... but does not couple the code to Spring.

#### Default initialization and destroy methods

When you write initialization and destroy method callbacks that do not use the Spring-specific InitializingBean and DisposableBean callback interfaces, you typically write methods with names such as init(), initialize(), dispose(), and so on. Ideally, the names of such lifecycle callback methods are standardized across a project so that all developers use the same method names and ensure consistency.

You can configure the Spring container to look for named initialization and destroy callback method names on every bean. This means that you, as an application developer, can write your application classes and use an initialization callback called init(), without having to configure an init-method="init" attribute with each bean definition. The Spring IoC container calls that method when the bean is created (and in accordance with the standard lifecycle callback contract described previously). This feature also enforces a consistent naming convention for initialization and destroy method callbacks.

Suppose that your initialization callback methods are named init() and destroy callback methods are named destroy(). Your class will resemble the class in the following example.

**public** **class** DefaultBlogService **implements** BlogService {

**private** BlogDao blogDao;

**public** **void** setBlogDao(BlogDao blogDao) {

**this**.blogDao = blogDao;

}

*// this is (unsurprisingly) the initialization callback method*

**public** **void** init() {

**if** (**this**.blogDao == null) {

**throw** **new** IllegalStateException("The [blogDao] property must be set.");

}

}

}

<beans default-init-method="init">

<bean id="blogService" class="com.foo.DefaultBlogService">

<property name="blogDao" ref="blogDao" />

</bean>

</beans>

The presence of the default-init-method attribute on the top-level <beans/> element attribute causes the Spring IoC container to recognize a method called init on beans as the initialization method callback. When a bean is created and assembled, if the bean class has such a method, it is invoked at the appropriate time.

You configure destroy method callbacks similarly (in XML, that is) by using the default-destroy-method attribute on the top-level <beans/> element.

Where existing bean classes already have callback methods that are named at variance with the convention, you can override the default by specifying (in XML, that is) the method name using the init-method and destroy-method attributes of the <bean/> itself.

The Spring container guarantees that a configured initialization callback is called immediately after a bean is supplied with all dependencies. Thus the initialization callback is called on the raw bean reference, which means that AOP interceptors and so forth are not yet applied to the bean. A target bean is fully created first, then an AOP proxy (for example) with its interceptor chain is applied. If the target bean and the proxy are defined separately, your code can even interact with the raw target bean, bypassing the proxy. Hence, it would be inconsistent to apply the interceptors to the init method, because doing so would couple the lifecycle of the target bean with its proxy/interceptors and leave strange semantics when your code interacts directly to the raw target bean.

#### Combining lifecycle mechanisms

As of Spring 2.5, you have three options for controlling bean lifecycle behavior: the [InitializingBean](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-lifecycle-initializingbean) and [DisposableBean](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-lifecycle-disposablebean) callback interfaces; custom init() and destroy() methods; and the [@PostConstruct and @PreDestroy annotations](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-postconstruct-and-predestroy-annotations). You can combine these mechanisms to control a given bean.

|  |
| --- |
| [Note] |
| If multiple lifecycle mechanisms are configured for a bean, and each mechanism is configured with a different method name, then each configured method is executed in the order listed below. However, if the same method name is configured - for example, init() for an initialization method - for more than one of these lifecycle mechanisms, that method is executed once, as explained in the preceding section. |

Multiple lifecycle mechanisms configured for the same bean, with different initialization methods, are called as follows:

* Methods annotated with @PostConstruct
* afterPropertiesSet() as defined by the InitializingBean callback interface
* A custom configured init() method

Destroy methods are called in the same order:

* Methods annotated with @PreDestroy
* destroy() as defined by the DisposableBean callback interface
* A custom configured destroy() method

#### Startup and shutdown callbacks

The Lifecycle interface defines the essential methods for any object that has its own lifecycle requirements (e.g. starts and stops some background process):

**public** **interface** Lifecycle {

**void** start();

**void** stop();

**boolean** isRunning();

}

Any Spring-managed object may implement that interface. Then, when the ApplicationContext itself starts and stops, it will cascade those calls to all Lifecycle implementations defined within that context. It does this by delegating to a LifecycleProcessor:

**public** **interface** LifecycleProcessor **extends** Lifecycle {

**void** onRefresh();

**void** onClose();

}

Notice that the LifecycleProcessor is itself an extension of the Lifecycle interface. It also adds two other methods for reacting to the context being refreshed and closed.

The order of startup and shutdown invocations can be important. If a "depends-on" relationship exists between any two objects, the dependent side will start after its dependency, and it will stop before its dependency. However, at times the direct dependencies are unknown. You may only know that objects of a certain type should start prior to objects of another type. In those cases, the SmartLifecycle interface defines another option, namely the getPhase() method as defined on its super-interface, Phased.

**public** **interface** Phased {

**int** getPhase();

}

**public** **interface** SmartLifecycle **extends** Lifecycle, Phased {

**boolean** isAutoStartup();

**void** stop(Runnable callback);

}

When starting, the objects with the lowest phase start first, and when stopping, the reverse order is followed. Therefore, an object that implements SmartLifecycle and whose getPhase() method returns Integer.MIN\_VALUE would be among the first to start and the last to stop. At the other end of the spectrum, a phase value of Integer.MAX\_VALUE would indicate that the object should be started last and stopped first (likely because it depends on other processes to be running). When considering the phase value, it's also important to know that the default phase for any "normal" Lifecycle object that does not implement SmartLifecycle would be 0. Therefore, any negative phase value would indicate that an object should start before those standard components (and stop after them), and vice versa for any positive phase value.

As you can see the stop method defined by SmartLifecycle accepts a callback. Any implementation must invoke that callback's run() method after that implementation's shutdown process is complete. That enables asynchronous shutdown where necessary since the default implementation of the LifecycleProcessor interface, DefaultLifecycleProcessor, will wait up to its timeout value for the group of objects within each phase to invoke that callback. The default per-phase timeout is 30 seconds. You can override the default lifecycle processor instance by defining a bean named "lifecycleProcessor" within the context. If you only want to modify the timeout, then defining the following would be sufficient:

<bean id="lifecycleProcessor" class="org.springframework.context.support.DefaultLifecycleProcessor">

*<!-- timeout value in milliseconds -->*

<property name="timeoutPerShutdownPhase" value="10000"/>

</bean>

As mentioned, the LifecycleProcessor interface defines callback methods for the refreshing and closing of the context as well. The latter will simply drive the shutdown process as if stop() had been called explicitly, but it will happen when the context is closing. The 'refresh' callback on the other hand enables another feature of SmartLifecycle beans. When the context is refreshed (after all objects have been instantiated and initialized), that callback will be invoked, and at that point the default lifecycle processor will check the boolean value returned by each SmartLifecycle object's isAutoStartup() method. If "true", then that object will be started at that point rather than waiting for an explicit invocation of the context's or its own start() method (unlike the context refresh, the context start does not happen automatically for a standard context implementation). The "phase" value as well as any "depends-on" relationships will determine the startup order in the same way as described above.

#### Shutting down the Spring IoC container gracefully in non-web applications

|  |
| --- |
| [Note] |
| This section applies only to non-web applications. Spring's web-based ApplicationContext implementations already have code in place to shut down the Spring IoC container gracefully when the relevant web application is shut down. |

If you are using Spring's IoC container in a non-web application environment; for example, in a rich client desktop environment; you register a shutdown hook with the JVM. Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released. Of course, you must still configure and implement these destroy callbacks correctly.

To register a shutdown hook, you call the registerShutdownHook() method that is declared on the AbstractApplicationContext class:

**import** org.springframework.context.support.AbstractApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **final** **class** Boot {

**public** **static** **void** main(**final** String[] args) **throws** Exception {

AbstractApplicationContext ctx

= **new** ClassPathXmlApplicationContext(**new** String []{"beans.xml"});

*// add a shutdown hook for the above context...*

ctx.registerShutdownHook();

*// app runs here...*

*// main method exits, hook is called prior to the app shutting down...*

}

}

### 5.6.2 ApplicationContextAware and BeanNameAware

When an ApplicationContext creates a class that implements the org.springframework.context.ApplicationContextAware interface, the class is provided with a reference to that ApplicationContext.

**public** **interface** ApplicationContextAware {

**void** setApplicationContext(ApplicationContext applicationContext) **throws** BeansException;

}

Thus beans can manipulate programmatically the ApplicationContext that created them, through the ApplicationContext interface, or by casting the reference to a known subclass of this interface, such as ConfigurableApplicationContext, which exposes additional functionality. One use would be the programmatic retrieval of other beans. Sometimes this capability is useful; however, in general you should avoid it, because it couples the code to Spring and does not follow the Inversion of Control style, where collaborators are provided to beans as properties. Other methods of the ApplicationContext provide access to file resources, publishing application events, and accessing a MessageSource. These additional features are described in [Section 5.14, “Additional Capabilities of the ApplicationContext”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#context-introduction)

As of Spring 2.5, autowiring is another alternative to obtain reference to the ApplicationContext. The "traditional" constructor and byType autowiring modes (as described in [Section 5.4.5, “Autowiring collaborators”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-autowire)) can provide a dependency of type ApplicationContext for a constructor argument or setter method parameter, respectively. For more flexibility, including the ability to autowire fields and multiple parameter methods, use the new annotation-based autowiring features. If you do, the ApplicationContext is autowired into a field, constructor argument, or method parameter that is expecting the ApplicationContext type if the field, constructor, or method in question carries the @Autowired annotation. For more information, see [Section 5.9.2, “@Autowired”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-autowired-annotation).

When an ApplicationContext creates a class that implements the org.springframework.beans.factory.BeanNameAware interface, the class is provided with a reference to the name defined in its associated object definition.

**public** **interface** BeanNameAware {

**void** setBeanName(string name) **throws** BeansException;

}

The callback is invoked after population of normal bean properties but before an initialization callback such as InitializingBeans afterPropertiesSet or a custom init-method.

### 5.6.3 Other Aware interfaces

Besides ApplicationContextAware and BeanNameAware discussed above, Spring offers a range of *Aware* interfaces that allow beans to indicate to the container that they require a certain infrastructure dependency. The most important Aware interfaces are summarized below - as a general rule, the name is a good indication of the dependency type:

**Table 5.4. Aware interfaces**

| **Name** | **Injected Dependency** | **Explained in...** |
| --- | --- | --- |
| ApplicationContextAware | Declaring ApplicationContext | [Section 5.6.2, “ApplicationContextAware and BeanNameAware”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-aware) |
| ApplicationEventPublisherAware | Event publisher of the enclosing ApplicationContext | [Section 5.14, “Additional Capabilities of the ApplicationContext”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#context-introduction) |
| BeanClassLoaderAware | Class loader used to load the bean classes. | [Section 5.3.2, “Instantiating beans”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-class) |
| BeanFactoryAware | Declaring BeanFactory | [Section 5.6.2, “ApplicationContextAware and BeanNameAware”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-aware) |
| BeanNameAware | Name of the declaring bean | [Section 5.6.2, “ApplicationContextAware and BeanNameAware”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-aware) |
| BootstrapContextAware | Resource adapter BootstrapContext the container runs in. Typically available only in JCA aware ApplicationContexts | [Chapter 25, *JCA CCI*](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/cci.html) |
| LoadTimeWeaverAware | Defined weaver for processing class definition at load time | [Section 9.8.4, “Load-time weaving with AspectJ in the Spring Framework”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/aop.html#aop-aj-ltw) |
| MessageSourceAware | Configured strategy for resolving messages (with support for parametrization and internationalization) | [Section 5.14, “Additional Capabilities of the ApplicationContext”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#context-introduction) |
| NotificationPublisherAware | Spring JMX notification publisher | [Section 24.7, “Notifications”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/jmx.html#jmx-notifications) |
| PortletConfigAware | Current PortletConfig the container runs in. Valid only in a web-aware Spring ApplicationContext | [Chapter 20, *Portlet MVC Framework*](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/portlet.html) |
| PortletContextAware | Current PortletContext the container runs in. Valid only in a web-aware Spring ApplicationContext | [Chapter 20, *Portlet MVC Framework*](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/portlet.html) |
| ResourceLoaderAware | Configured loader for low-level access to resources | [Chapter 6, *Resources*](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/resources.html) |
| ServletConfigAware | Current ServletConfig the container runs in. Valid only in a web-aware Spring ApplicationContext | [Chapter 17, *Web MVC framework*](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/mvc.html) |
| ServletContextAware | Current ServletContext the container runs in. Valid only in a web-aware Spring ApplicationContext | [Chapter 17, *Web MVC framework*](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/mvc.html) |

Note again that usage of these interfaces ties your code to the Spring API and does not follow the Inversion of Control style. As such, they are recommended for infrastructure beans that require programmatic access to the container.

## 5.7 Bean definition inheritance

A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information such as initialization method, static factory method name, and so on. A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed. Using parent and child bean definitions can save a lot of typing. Effectively, this is a form of templating.

If you work with an ApplicationContext interface programmatically, child bean definitions are represented by the ChildBeanDefinition class. Most users do not work with them on this level, instead configuring bean definitions declaratively in something like the ClassPathXmlApplicationContext. When you use XML-based configuration metadata, you indicate a child bean definition by using the parent attribute, specifying the parent bean as the value of this attribute.

<bean id="inheritedTestBean" abstract="true"

class="org.springframework.beans.TestBean">

<property name="name" value="parent"/>

<property name="age" value="1"/>

</bean>

<bean id="inheritsWithDifferentClass"

class="org.springframework.beans.DerivedTestBean"

**parent="inheritedTestBean"** init-method="initialize">

<property name="name" value="override"/>

*<!-- the age property value of 1 will be inherited from parent -->*

</bean>

A child bean definition uses the bean class from the parent definition if none is specified, but can also override it. In the latter case, the child bean class must be compatible with the parent, that is, it must accept the parent's property values.

A child bean definition inherits scope, constructor argument values, property values, and method overrides from the parent, with the option to add new values. Any scope, initialization method, destroy method, and/or static factory method settings that you specify will override the corresponding parent settings.

The remaining settings are always taken from the child definition: depends on, autowire mode, dependency check, singleton, lazy init.

The preceding example explicitly marks the parent bean definition as abstract by using the abstract attribute. If the parent definition does not specify a class, explicitly marking the parent bean definition as abstract is required, as follows:

<bean id="inheritedTestBeanWithoutClass" abstract="true">

<property name="name" value="parent"/>

<property name="age" value="1"/>

</bean>

<bean id="inheritsWithClass" class="org.springframework.beans.DerivedTestBean"

parent="inheritedTestBeanWithoutClass" init-method="initialize">

<property name="name" value="override"/>

*<!-- age will inherit the value of 1 from the parent bean definition-->*

</bean>

The parent bean cannot be instantiated on its own because it is incomplete, and it is also explicitly marked as abstract. When a definition is abstract like this, it is usable only as a pure template bean definition that serves as a parent definition for child definitions. Trying to use such an abstract parent bean on its own, by referring to it as a ref property of another bean or doing an explicit getBean() call with the parent bean id, returns an error. Similarly, the container's internal preInstantiateSingletons() method ignores bean definitions that are defined as abstract.

|  |
| --- |
| [Note] |
| ApplicationContext pre-instantiates all singletons by default. Therefore, it is important (at least for singleton beans) that if you have a (parent) bean definition which you intend to use only as a template, and this definition specifies a class, you must make sure to set the abstract attribute to true, otherwise the application context will actually (attempt to) pre-instantiate the abstract bean. |

## 5.8 Container Extension Points

Typically, an application developer does not need to subclass ApplicationContext implementation classes. Instead, the Spring IoC container can be extended by plugging in implementations of special integration interfaces. The next few sections describe these integration interfaces.

### 5.8.1 Customizing beans using a BeanPostProcessor

The BeanPostProcessor interface defines callback methods that you can implement to provide your own (or override the container's default) instantiation logic, dependency-resolution logic, and so forth. If you want to implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean, you can plug in one or more BeanPostProcessor implementations.

You can configure multiple BeanPostProcessor instances, and you can control the order in which these BeanPostProcessors execute by setting the order property. You can set this property only if the BeanPostProcessor implements the Ordered interface; if you write your own BeanPostProcessor you should consider implementing the Ordered interface too. For further details, consult the Javadoc for the BeanPostProcessor and Ordered interfaces. See also the note below on [programmatic registration of BeanPostProcessors](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-programmatically-registering-beanpostprocessors)

|  |
| --- |
| [Note] |
| BeanPostProcessors operate on bean (or object) instances; that is to say, the Spring IoC container instantiates a bean instance and then BeanPostProcessors do their work.  BeanPostProcessors are scoped per-container. This is only relevant if you are using container hierarchies. If you define a BeanPostProcessor in one container, it will only post-process the beans in that container. In other words, beans that are defined in one container are not post-processed by a BeanPostProcessor defined in another container, even if both containers are part of the same hierarchy.  To change the actual bean definition (i.e., the blueprint that defines the bean), you instead need to use a BeanFactoryPostProcessor as described in [Section 5.8.2, “Customizing configuration metadata with a BeanFactoryPostProcessor”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-extension-factory-postprocessors). |

The org.springframework.beans.factory.config.BeanPostProcessor interface consists of exactly two callback methods. When such a class is registered as a post-processor with the container, for each bean instance that is created by the container, the post-processor gets a callback from the container both before container initialization methods (such as InitializingBean's afterPropertiesSet() and any declared init method) are called as well as after any bean initialization callbacks. The post-processor can take any action with the bean instance, including ignoring the callback completely. A bean post-processor typically checks for callback interfaces or may wrap a bean with a proxy. Some Spring AOP infrastructure classes are implemented as bean post-processors in order to provide proxy-wrapping logic.

An ApplicationContext automatically detects any beans that are defined in the configuration metadata which implement the BeanPostProcessor interface. The ApplicationContext registers these beans as post-processors so that they can be called later upon bean creation. Bean post-processors can be deployed in the container just like any other beans.

|  |
| --- |
| [Note] |
| While the recommended approach for BeanPostProcessor registration is through ApplicationContext auto-detection (as described above), it is also possible to register them programmatically against a ConfigurableBeanFactory using the addBeanPostProcessor method. This can be useful when needing to evaluate conditional logic before registration, or even for copying bean post processors across contexts in a hierarchy. Note however that BeanPostProcessors added programmatically do not respect the *Ordered*interface. Here it is the order of registration that dictates the order of execution. Note also that BeanPostProcessors registered programmatically are always processed before those registered through auto-detection, regardless of any explicit ordering. |
| [Note] |
| Classes that implement the BeanPostProcessor interface are special and are treated differently by the container. All BeanPostProcessors and beans that they reference directly are instantiated on startup, as part of the special startup phase of the ApplicationContext. Next, all BeanPostProcessors are registered in a sorted fashion and applied to all further beans in the container. Because AOP auto-proxying is implemented as a BeanPostProcessor itself, neither BeanPostProcessors nor the beans they reference directly are eligible for auto-proxying, and thus do not have aspects woven into them.  For any such bean, you should see an informational log message: “Bean foo is not eligible for getting processed by all BeanPostProcessor interfaces (for example: not eligible for auto-proxying)”.  Note that if you have beans wired into your BeanPostProcessor using autowiring or @Resource (which may fall back to autowiring), Spring might access unexpected beans when searching for type-matching dependency candidates, and therefore make them ineligible for auto-proxying or other kinds of bean post-processing. For example, if you have a dependency annotated with @Resource where the field/setter name does not directly correspond to the declared name of a bean and no name attribute is used, then Spring will access other beans for matching them by type. | |

The following examples show how to write, register, and use BeanPostProcessors in an ApplicationContext.

#### Example: Hello World, BeanPostProcessor-style

This first example illustrates basic usage. The example shows a custom BeanPostProcessor implementation that invokes the toString() method of each bean as it is created by the container and prints the resulting string to the system console.

Find below the custom BeanPostProcessor implementation class definition:

**package** scripting;

**import** org.springframework.beans.factory.config.BeanPostProcessor;

**import** org.springframework.beans.BeansException;

**public** **class** InstantiationTracingBeanPostProcessor **implements** BeanPostProcessor {

*// simply return the instantiated bean as-is*

**public** Object postProcessBeforeInitialization(Object bean, String beanName)

**throws** BeansException {

**return** bean; *// we could potentially return any object reference here...*

}

**public** Object postProcessAfterInitialization(Object bean, String beanName)

**throws** BeansException {

System.out.println("Bean '" + beanName + "' created : " + bean.toString());

**return** bean;

}

}

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:lang="http://www.springframework.org/schema/lang"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/lang

http://www.springframework.org/schema/lang/spring-lang.xsd">

<lang:groovy id="messenger"

script-source="classpath:org/springframework/scripting/groovy/Messenger.groovy">

<lang:property name="message" value="Fiona Apple Is Just So Dreamy."/>

</lang:groovy>

*<!--*

*when the above bean (messenger) is instantiated, this custom*

*BeanPostProcessor implementation will output the fact to the system console*

*-->*

<bean class="scripting.InstantiationTracingBeanPostProcessor"/>

</beans>

Notice how the InstantiationTracingBeanPostProcessor is simply defined. It does not even have a name, and because it is a bean it can be dependency-injected just like any other bean. (The preceding configuration also defines a bean that is backed by a Groovy script. The Spring 2.0 dynamic language support is detailed in the chapter entitled [Chapter 28, *Dynamic language support*](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/dynamic-language.html).)

The following simple Java application executes the preceding code and configuration:

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**import** org.springframework.scripting.Messenger;

**public** **final** **class** Boot {

**public** **static** **void** main(**final** String[] args) **throws** Exception {

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("scripting/beans.xml");

Messenger messenger = (Messenger) ctx.getBean("messenger");

System.out.println(messenger);

}

}

The output of the preceding application resembles the following:

Bean 'messenger' created : org.springframework.scripting.groovy.GroovyMessenger@272961

org.springframework.scripting.groovy.GroovyMessenger@272961

#### Example: The RequiredAnnotationBeanPostProcessor

Using callback interfaces or annotations in conjunction with a custom BeanPostProcessor implementation is a common means of extending the Spring IoC container. An example is Spring's RequiredAnnotationBeanPostProcessor — a BeanPostProcessor implementation that ships with the Spring distribution which ensures that JavaBean properties on beans that are marked with an (arbitrary) annotation are actually (configured to be) dependency-injected with a value.

### 5.8.2 Customizing configuration metadata with a BeanFactoryPostProcessor

The next extension point that we will look at is the org.springframework.beans.factory.config.BeanFactoryPostProcessor. The semantics of this interface are similar to those of the BeanPostProcessor, with one major difference: BeanFactoryPostProcessors operate on the bean configuration metadata; that is, the Spring IoC container allows BeanFactoryPostProcessors to read the configuration metadata and potentially change it before the container instantiates any beans other than BeanFactoryPostProcessors.

You can configure multiple BeanFactoryPostProcessors, and you can control the order in which these BeanFactoryPostProcessors execute by setting the order property. However, you can only set this property if the BeanFactoryPostProcessor implements the Ordered interface. If you write your own BeanFactoryPostProcessor, you should consider implementing the Ordered interface too. Consult the Javadoc for the BeanFactoryPostProcessor and Ordered interfaces for more details.

|  |
| --- |
| [Note] |
| If you want to change the actual bean instances (i.e., the objects that are created from the configuration metadata), then you instead need to use a BeanPostProcessor (described above in [Section 5.8.1, “Customizing beans using a BeanPostProcessor”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-extension-bpp)). While it is technically possible to work with bean instances within a BeanFactoryPostProcessor (e.g., using BeanFactory.getBean()), doing so causes premature bean instantiation, violating the standard container lifecycle. This may cause negative side effects such as bypassing bean post processing.  Also, BeanFactoryPostProcessors are scoped per-container. This is only relevant if you are using container hierarchies. If you define a BeanFactoryPostProcessor in one container, it will only be applied to the bean definitions in that container. Bean definitions in one container will not be post-processed by BeanFactoryPostProcessors in another container, even if both containers are part of the same hierarchy. |

A bean factory post-processor is executed automatically when it is declared inside an ApplicationContext, in order to apply changes to the configuration metadata that define the container. Spring includes a number of predefined bean factory post-processors, such as PropertyOverrideConfigurer and PropertyPlaceholderConfigurer. A custom BeanFactoryPostProcessor can also be used, for example, to register custom property editors.

An ApplicationContext automatically detects any beans that are deployed into it that implement the BeanFactoryPostProcessor interface. It uses these beans as bean factory post-processors, at the appropriate time. You can deploy these post-processor beans as you would any other bean.

|  |
| --- |
| [Note] |
| As with BeanPostProcessors, you typically do not want to configure BeanFactoryPostProcessors for lazy initialization. If no other bean references a Bean(Factory)PostProcessor, that post-processor will not get instantiated at all. Thus, marking it for lazy initialization will be ignored, and the Bean(Factory)PostProcessor will be instantiated eagerly even if you set the default-lazy-init attribute to true on the declaration of your <beans /> element. |

#### Example: the PropertyPlaceholderConfigurer

You use the PropertyPlaceholderConfigurer to externalize property values from a bean definition in a separate file using the standard Java Properties format. Doing so enables the person deploying an application to customize environment-specific properties such as database URLs and passwords, without the complexity or risk of modifying the main XML definition file or files for the container.

Consider the following XML-based configuration metadata fragment, where a DataSource with placeholder values is defined. The example shows properties configured from an external Properties file. At runtime, a PropertyPlaceholderConfigurer is applied to the metadata that will replace some properties of the DataSource. The values to replace are specified as placeholders of the form ${property-name} which follows the Ant / log4j / JSP EL style.

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations" value="classpath:com/foo/jdbc.properties"/>

</bean>

<bean id="dataSource" destroy-method="close"

class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName" value="${jdbc.driverClassName}"/>

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

The actual values come from another file in the standard Java Properties format:

jdbc.driverClassName=org.hsqldb.jdbcDriver

jdbc.url=jdbc:hsqldb:hsql://production:9002

jdbc.username=sa

jdbc.password=root

Therefore, the string ${jdbc.username} is replaced at runtime with the value 'sa', and the same applies for other placeholder values that match keys in the properties file. The PropertyPlaceholderConfigurer checks for placeholders in most properties and attributes of a bean definition. Furthermore, the placeholder prefix and suffix can be customized.

With the context namespace introduced in Spring 2.5, it is possible to configure property placeholders with a dedicated configuration element. One or more locations can be provided as a comma-separated list in the location attribute.

<context:property-placeholder location="classpath:com/foo/jdbc.properties"/>

The PropertyPlaceholderConfigurer not only looks for properties in the Properties file you specify. By default it also checks against the Java System properties if it cannot find a property in the specified properties files. You can customize this behavior by setting the systemPropertiesMode property of the configurer with one of the following three supported integer values:

* never (0): Never check system properties
* fallback (1): Check system properties if not resolvable in the specified properties files. This is the default.
* override (2): Check system properties first, before trying the specified properties files. This allows system properties to override any other property source.

Consult the Javadoc for the PropertyPlaceholderConfigurer for more information.

|  |
| --- |
| [Tip] |
| You can use the PropertyPlaceholderConfigurer to substitute class names, which is sometimes useful when you have to pick a particular implementation class at runtime. For example:  <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  <property name="locations">  <value>classpath:com/foo/strategy.properties</value>  </property>  <property name="properties">  <value>custom.strategy.class=com.foo.DefaultStrategy</value>  </property>  </bean>  <bean id="serviceStrategy" class="${custom.strategy.class}"/>  If the class cannot be resolved at runtime to a valid class, resolution of the bean fails when it is about to be created, which is during the preInstantiateSingletons() phase of an ApplicationContext for a non-lazy-init bean. |

#### Example: the PropertyOverrideConfigurer

The PropertyOverrideConfigurer, another bean factory post-processor, resembles the PropertyPlaceholderConfigurer, but unlike the latter, the original definitions can have default values or no values at all for bean properties. If an overriding Properties file does not have an entry for a certain bean property, the default context definition is used.

Note that the bean definition is not aware of being overridden, so it is not immediately obvious from the XML definition file that the override configurer is being used. In case of multiple PropertyOverrideConfigurer instances that define different values for the same bean property, the last one wins, due to the overriding mechanism.

Properties file configuration lines take this format:

beanName.property=value

For example:

dataSource.driverClassName=com.mysql.jdbc.Driver

dataSource.url=jdbc:mysql:mydb

This example file can be used with a container definition that contains a bean called dataSource, which has driver and url properties.

Compound property names are also supported, as long as every component of the path except the final property being overridden is already non-null (presumably initialized by the constructors). In this example...

foo.fred.bob.sammy=123

... the sammy property of the bob property of the fred property of the foo bean is set to the scalar value 123.

|  |
| --- |
| [Note] |
| Specified override values are always literal values; they are not translated into bean references. This convention also applies when the original value in the XML bean definition specifies a bean reference. |

With the context namespace introduced in Spring 2.5, it is possible to configure property overriding with a dedicated configuration element:

<context:property-override location="classpath:override.properties"/>

### 5.8.3 Customizing instantiation logic with a FactoryBean

Implement the org.springframework.beans.factory.FactoryBean interface for objects that are themselves factories.

The FactoryBean interface is a point of pluggability into the Spring IoC container's instantiation logic. If you have complex initialization code that is better expressed in Java as opposed to a (potentially) verbose amount of XML, you can create your own FactoryBean, write the complex initialization inside that class, and then plug your custom FactoryBean into the container.

The FactoryBean interface provides three methods:

* Object getObject(): returns an instance of the object this factory creates. The instance can possibly be shared, depending on whether this factory returns singletons or prototypes.
* boolean isSingleton(): returns true if this FactoryBean returns singletons, false otherwise.
* Class getObjectType(): returns the object type returned by the getObject() method or null if the type is not known in advance.

The FactoryBean concept and interface is used in a number of places within the Spring Framework; more than 50 implementations of the FactoryBean interface ship with Spring itself.

When you need to ask a container for an actual FactoryBean instance itself instead of the bean it produces, preface the bean's id with the ampersand symbol (&) when calling the getBean() method of the ApplicationContext. So for a given FactoryBean with an id of myBean, invoking getBean("myBean") on the container returns the product of the FactoryBean; whereas, invoking getBean("&myBean") returns the FactoryBean instance itself.

## 5.9 Annotation-based container configuration

**Are annotations better than XML for configuring Spring?**

The introduction of annotation-based configurations raised the question of whether this approach is 'better' than XML. The short answer is it depends. The long answer is that each approach has its pros and cons, and usually it is up to the developer to decide which strategy suits her better. Due to the way they are defined, annotations provide a lot of context in their declaration, leading to shorter and more concise configuration. However, XML excels at wiring up components without touching their source code or recompiling them. Some developers prefer having the wiring close to the source while others argue that annotated classes are no longer POJOs and, furthermore, that the configuration becomes decentralized and harder to control.

No matter the choice, Spring can accommodate both styles and even mix them together. It's worth pointing out that through its [JavaConfig](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-java) option, Spring allows annotations to be used in a non-invasive way, without touching the target components source code and that in terms of tooling, all configuration styles are supported by the [SpringSource Tool Suite](http://www.springsource.com/products/sts).

An alternative to XML setups is provided by annotation-based configuration which rely on the bytecode metadata for wiring up components instead of angle-bracket declarations. Instead of using XML to describe a bean wiring, the developer moves the configuration into the component class itself by using annotations on the relevant class, method, or field declaration. As mentioned in [the section called “Example: The RequiredAnnotationBeanPostProcessor”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-extension-bpp-examples-rabpp), using a BeanPostProcessor in conjunction with annotations is a common means of extending the Spring IoC container. For example, Spring 2.0 introduced the possibility of enforcing required properties with the [@Required](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-required-annotation) annotation. Spring 2.5 made it possible to follow that same general approach to drive Spring's dependency injection. Essentially, the @Autowired annotation provides the same capabilities as described in [Section 5.4.5, “Autowiring collaborators”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-autowire) but with more fine-grained control and wider applicability. Spring 2.5 also added support for JSR-250 annotations such as @PostConstruct, and @PreDestroy. Spring 3.0 added support for JSR-330 (Dependency Injection for Java) annotations contained in the javax.inject package such as @Inject and @Named. Details about those annotations can be found in the [relevant section](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-standard-annotations).

|  |
| --- |
| [Note] |
| Annotation injection is performed before XML injection, thus the latter configuration will override the former for properties wired through both approaches. |

As always, you can register them as individual bean definitions, but they can also be implicitly registered by including the following tag in an XML-based Spring configuration (notice the inclusion of the context namespace):

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

*xmlns:context="http://www.springframework.org/schema/context"*

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

*<context:annotation-config/>*

</beans>

(The implicitly registered post-processors include [AutowiredAnnotationBeanPostProcessor](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/AutowiredAnnotationBeanPostProcessor.html), [CommonAnnotationBeanPostProcessor](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/CommonAnnotationBeanPostProcessor.html), [PersistenceAnnotationBeanPostProcessor](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/orm/jpa/support/PersistenceAnnotationBeanPostProcessor.html), as well as the aforementioned [RequiredAnnotationBeanPostProcessor](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/RequiredAnnotationBeanPostProcessor.html).)

|  |
| --- |
| [Note] |
| <context:annotation-config/> only looks for annotations on beans in the same application context in which it is defined. This means that, if you put <context:annotation-config/> in a WebApplicationContext for a DispatcherServlet, it only checks for @Autowired beans in your controllers, and not your services. See [Section 17.2, “The DispatcherServlet”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/mvc.html#mvc-servlet) for more information. |

### 5.9.1 @Required

The @Required annotation applies to bean property setter methods, as in the following example:

**public** **class** SimpleMovieLister {

**private** MovieFinder movieFinder;

*@Required*

**public** **void** setMovieFinder(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

*// ...*

}

This annotation simply indicates that the affected bean property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring. The container throws an exception if the affected bean property has not been populated; this allows for eager and explicit failure, avoiding NullPointerExceptions or the like later on. It is still recommended that you put assertions into the bean class itself, for example, into an init method. Doing so enforces those required references and values even when you use the class outside of a container.

### 5.9.2 @Autowired

As expected, you can apply the @Autowired annotation to "traditional" setter methods:

**public** **class** SimpleMovieLister {

**private** MovieFinder movieFinder;

*@Autowired*

**public** **void** setMovieFinder(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

*// ...*

}

|  |
| --- |
| [Note] |
| JSR 330's @Inject annotation can be used in place of Spring's @Autowired annotation in the examples below. See [here](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-standard-annotations) for more details |

You can also apply the annotation to methods with arbitrary names and/or multiple arguments:

**public** **class** MovieRecommender {

**private** MovieCatalog movieCatalog;

**private** CustomerPreferenceDao customerPreferenceDao;

*@Autowired*

**public** **void** prepare(MovieCatalog movieCatalog,

CustomerPreferenceDao customerPreferenceDao) {

**this**.movieCatalog = movieCatalog;

**this**.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

You can apply @Autowired to constructors and fields:

**public** **class** MovieRecommender {

*@Autowired*

**private** MovieCatalog movieCatalog;

**private** CustomerPreferenceDao customerPreferenceDao;

*@Autowired*

**public** MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {

**this**.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

It is also possible to provide all beans of a particular type from the ApplicationContext by adding the annotation to a field or method that expects an array of that type:

**public** **class** MovieRecommender {

*@Autowired*

**private** MovieCatalog[] movieCatalogs;

*// ...*

}

The same applies for typed collections:

**public** **class** MovieRecommender {

**private** Set<MovieCatalog> movieCatalogs;

*@Autowired*

**public** **void** setMovieCatalogs(Set<MovieCatalog> movieCatalogs) {

**this**.movieCatalogs = movieCatalogs;

}

*// ...*

}

Even typed Maps can be autowired as long as the expected key type is String. The Map values will contain all beans of the expected type, and the keys will contain the corresponding bean names:

**public** **class** MovieRecommender {

**private** Map<String, MovieCatalog> movieCatalogs;

*@Autowired*

**public** **void** setMovieCatalogs(Map<String, MovieCatalog> movieCatalogs) {

**this**.movieCatalogs = movieCatalogs;

}

*// ...*

}

By default, the autowiring fails whenever zero candidate beans are available; the default behavior is to treat annotated methods, constructors, and fields as indicating required dependencies. This behavior can be changed as demonstrated below.

**public** **class** SimpleMovieLister {

**private** MovieFinder movieFinder;

*@Autowired(required=false)*

**public** **void** setMovieFinder(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

*// ...*

}

|  |
| --- |
| [Note] |
| Only one annotated constructor per-class can be marked as required, but multiple non-required constructors can be annotated. In that case, each is considered among the candidates and Spring uses the greediest constructor whose dependencies can be satisfied, that is the constructor that has the largest number of arguments.  @Autowired's required attribute is recommended over the @Required annotation. The required attribute indicates that the property is not required for autowiring purposes, the property is ignored if it cannot be autowired. @Required, on the other hand, is stronger in that it enforces the property that was set by any means supported by the container. If no value is injected, a corresponding exception is raised. |

You can also use @Autowired for interfaces that are well-known resolvable dependencies: BeanFactory, ApplicationContext, Environment, ResourceLoader, ApplicationEventPublisher, and MessageSource. These interfaces and their extended interfaces, such as ConfigurableApplicationContext or ResourcePatternResolver, are automatically resolved, with no special setup necessary.

**public** **class** MovieRecommender {

*@Autowired*

**private** ApplicationContext context;

**public** MovieRecommender() {

}

*// ...*

}

|  |
| --- |
| [Note] |
| @Autowired, @Inject, @Resource, and @Value annotations are handled by a Spring BeanPostProcessor implementations which in turn means that you cannot apply these annotations within your own BeanPostProcessor or BeanFactoryPostProcessor types (if any). These types must be 'wired up' explicitly via XML or using a Spring @Bean method. |

### 5.9.3 Fine-tuning annotation-based autowiring with qualifiers

Because autowiring by type may lead to multiple candidates, it is often necessary to have more control over the selection process. One way to accomplish this is with Spring's @Qualifier annotation. You can associate qualifier values with specific arguments, narrowing the set of type matches so that a specific bean is chosen for each argument. In the simplest case, this can be a plain descriptive value:

**public** **class** MovieRecommender {

*@Autowired*

**@Qualifier("main")**

**private** MovieCatalog movieCatalog;

*// ...*

}

The @Qualifier annotation can also be specified on individual constructor arguments or method parameters:

**public** **class** MovieRecommender {

**private** MovieCatalog movieCatalog;

**private** CustomerPreferenceDao customerPreferenceDao;

*@Autowired*

**public** **void** prepare(**@Qualifier("main")** MovieCatalog movieCatalog,

CustomerPreferenceDao customerPreferenceDao) {

**this**.movieCatalog = movieCatalog;

**this**.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

The corresponding bean definitions appear as follows. The bean with qualifier value "main" is wired with the constructor argument that is qualified with the same value.

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

<context:annotation-config/>

<bean class="example.SimpleMovieCatalog">

**<qualifier value="main"/>**

*<!-- inject any dependencies required by this bean -->*

</bean>

<bean class="example.SimpleMovieCatalog">

**<qualifier value="action"/>**

*<!-- inject any dependencies required by this bean -->*

</bean>

<bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>

For a fallback match, the bean name is considered a default qualifier value. Thus you can define the bean with an id "main" instead of the nested qualifier element, leading to the same matching result. However, although you can use this convention to refer to specific beans by name, @Autowired is fundamentally about type-driven injection with optional semantic qualifiers. This means that qualifier values, even with the bean name fallback, always have narrowing semantics within the set of type matches; they do not semantically express a reference to a unique bean id. Good qualifier values are "main" or "EMEA" or "persistent", expressing characteristics of a specific component that are independent from the bean id, which may be auto-generated in case of an anonymous bean definition like the one in the preceding example.

Qualifiers also apply to typed collections, as discussed above, for example, to Set<MovieCatalog>. In this case, all matching beans according to the declared qualifiers are injected as a collection. This implies that qualifiers do not have to be unique; they rather simply constitute filtering criteria. For example, you can define multiple MovieCatalog beans with the same qualifier value "action"; all of which would be injected into a Set<MovieCatalog> annotated with @Qualifier("action").

|  |
| --- |
| [Tip] |
| If you intend to express annotation-driven injection by name, do not primarily use @Autowired, even if is technically capable of referring to a bean name through @Qualifier values. Instead, use the JSR-250 @Resource annotation, which is semantically defined to identify a specific target component by its unique name, with the declared type being irrelevant for the matching process.  As a specific consequence of this semantic difference, beans that are themselves defined as a collection or map type cannot be injected through @Autowired, because type matching is not properly applicable to them. Use @Resource for such beans, referring to the specific collection or map bean by unique name.  @Autowired applies to fields, constructors, and multi-argument methods, allowing for narrowing through qualifier annotations at the parameter level. By contrast, @Resource is supported only for fields and bean property setter methods with a single argument. As a consequence, stick with qualifiers if your injection target is a constructor or a multi-argument method. |

You can create your own custom qualifier annotations. Simply define an annotation and provide the @Qualifier annotation within your definition:

*@Target({ElementType.FIELD, ElementType.PARAMETER})*

*@Retention(RetentionPolicy.RUNTIME)*

**@Qualifier**

**public** *@interface* Genre {

String value();

}

Then you can provide the custom qualifier on autowired fields and parameters:

**public** **class** MovieRecommender {

*@Autowired*

**@Genre("Action")**

**private** MovieCatalog actionCatalog;

**private** MovieCatalog comedyCatalog;

*@Autowired*

**public** **void** setComedyCatalog(**@Genre("Comedy")** MovieCatalog comedyCatalog) {

**this**.comedyCatalog = comedyCatalog;

}

*// ...*

}

Next, provide the information for the candidate bean definitions. You can add <qualifier/> tags as sub-elements of the <bean/> tag and then specify the type and value to match your custom qualifier annotations. The type is matched against the fully-qualified class name of the annotation. Or, as a convenience if no risk of conflicting names exists, you can use the short class name. Both approaches are demonstrated in the following example.

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

<context:annotation-config/>

<bean class="example.SimpleMovieCatalog">

**<qualifier type="Genre" value="Action"/>**

*<!-- inject any dependencies required by this bean -->*

</bean>

<bean class="example.SimpleMovieCatalog">

**<qualifier type="example.Genre" value="Comedy"/>**

*<!-- inject any dependencies required by this bean -->*

</bean>

<bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>

In [Section 5.10, “Classpath scanning and managed components”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-classpath-scanning), you will see an annotation-based alternative to providing the qualifier metadata in XML. Specifically, see [Section 5.10.7, “Providing qualifier metadata with annotations”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-scanning-qualifiers).

In some cases, it may be sufficient to use an annotation without a value. This may be useful when the annotation serves a more generic purpose and can be applied across several different types of dependencies. For example, you may provide an offline catalog that would be searched when no Internet connection is available. First define the simple annotation:

*@Target({ElementType.FIELD, ElementType.PARAMETER})*

*@Retention(RetentionPolicy.RUNTIME)*

*@Qualifier*

**public** *@interface* Offline {

}

Then add the annotation to the field or property to be autowired:

**public** **class** MovieRecommender {

*@Autowired*

**@Offline**

**private** MovieCatalog offlineCatalog;

*// ...*

}

Now the bean definition only needs a qualifier type:

<bean class="example.SimpleMovieCatalog">

**<qualifier type="Offline"/>**

*<!-- inject any dependencies required by this bean -->*

</bean>

You can also define custom qualifier annotations that accept named attributes in addition to or instead of the simple value attribute. If multiple attribute values are then specified on a field or parameter to be autowired, a bean definition must match all such attribute values to be considered an autowire candidate. As an example, consider the following annotation definition:

*@Target({ElementType.FIELD, ElementType.PARAMETER})*

*@Retention(RetentionPolicy.RUNTIME)*

*@Qualifier*

**public** *@interface* MovieQualifier {

String genre();

Format format();

}

In this case Format is an enum:

**public** enum Format {

VHS, DVD, BLURAY

}

The fields to be autowired are annotated with the custom qualifier and include values for both attributes: genre and format.

**public** **class** MovieRecommender {

*@Autowired*

*@MovieQualifier(format=Format.VHS, genre="Action")*

**private** MovieCatalog actionVhsCatalog;

*@Autowired*

*@MovieQualifier(format=Format.VHS, genre="Comedy")*

**private** MovieCatalog comedyVhsCatalog;

*@Autowired*

*@MovieQualifier(format=Format.DVD, genre="Action")*

**private** MovieCatalog actionDvdCatalog;

*@Autowired*

*@MovieQualifier(format=Format.BLURAY, genre="Comedy")*

**private** MovieCatalog comedyBluRayCatalog;

*// ...*

}

Finally, the bean definitions should contain matching qualifier values. This example also demonstrates that bean meta attributes may be used instead of the <qualifier/> sub-elements. If available, the <qualifier/> and its attributes take precedence, but the autowiring mechanism falls back on the values provided within the <meta/> tags if no such qualifier is present, as in the last two bean definitions in the following example.

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

<context:annotation-config/>

<bean class="example.SimpleMovieCatalog">

<qualifier type="MovieQualifier">

<attribute key="format" value="VHS"/>

<attribute key="genre" value="Action"/>

</qualifier>

*<!-- inject any dependencies required by this bean -->*

</bean>

<bean class="example.SimpleMovieCatalog">

<qualifier type="MovieQualifier">

<attribute key="format" value="VHS"/>

<attribute key="genre" value="Comedy"/>

</qualifier>

*<!-- inject any dependencies required by this bean -->*

</bean>

<bean class="example.SimpleMovieCatalog">

<meta key="format" value="DVD"/>

<meta key="genre" value="Action"/>

*<!-- inject any dependencies required by this bean -->*

</bean>

<bean class="example.SimpleMovieCatalog">

<meta key="format" value="BLURAY"/>

<meta key="genre" value="Comedy"/>

*<!-- inject any dependencies required by this bean -->*

</bean>

</beans>

### 5.9.4 CustomAutowireConfigurer

The [CustomAutowireConfigurer](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/CustomAutowireConfigurer.html) is a BeanFactoryPostProcessor that enables you to register your own custom qualifier annotation types even if they are not annotated with Spring's @Qualifier annotation.

<bean id="customAutowireConfigurer"

class="org.springframework.beans.factory.annotation.CustomAutowireConfigurer">

<property name="customQualifierTypes">

<set>

<value>example.CustomQualifier</value>

</set>

</property>

</bean>

The particular implementation of AutowireCandidateResolver that is activated for the application context depends on the Java version. In versions earlier than Java 5, the qualifier annotations are not supported, and therefore autowire candidates are solely determined by the autowire-candidate value of each bean definition as well as by any default-autowire-candidates pattern(s) available on the <beans/> element. In Java 5 or later, the presence of @Qualifier annotations and any custom annotations registered with the CustomAutowireConfigurer will also play a role.

Regardless of the Java version, when multiple beans qualify as autowire candidates, the determination of a "primary" candidate is the same: if exactly one bean definition among the candidates has a primary attribute set to true, it will be selected.

### 5.9.5 @Resource

Spring also supports injection using the JSR-250 @Resource annotation on fields or bean property setter methods. This is a common pattern in Java EE 5 and 6, for example in JSF 1.2 managed beans or JAX-WS 2.0 endpoints. Spring supports this pattern for Spring-managed objects as well.

@Resource takes a name attribute, and by default Spring interprets that value as the bean name to be injected. In other words, it follows by-name semantics, as demonstrated in this example:

**public** **class** SimpleMovieLister {

**private** MovieFinder movieFinder;

**@Resource(name="myMovieFinder")**

**public** **void** setMovieFinder(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

}

If no name is specified explicitly, the default name is derived from the field name or setter method. In case of a field, it takes the field name; in case of a setter method, it takes the bean property name. So the following example is going to have the bean with name "movieFinder" injected into its setter method:

**public** **class** SimpleMovieLister {

**private** MovieFinder movieFinder;

**@Resource**

**public** **void** setMovieFinder(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

}

|  |
| --- |
| [Note] |
| The name provided with the annotation is resolved as a bean name by the ApplicationContext of which the CommonAnnotationBeanPostProcessor is aware. The names can be resolved through JNDI if you configure Spring's [SimpleJndiBeanFactory](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/jndi/support/SimpleJndiBeanFactory.html) explicitly. However, it is recommended that you rely on the default behavior and simply use Spring's JNDI lookup capabilities to preserve the level of indirection. |

In the exclusive case of @Resource usage with no explicit name specified, and similar to @Autowired, @Resource finds a primary type match instead of a specific named bean and resolves well-known resolvable dependencies: the BeanFactory, ApplicationContext, ResourceLoader, ApplicationEventPublisher, and MessageSource interfaces.

Thus in the following example, the customerPreferenceDao field first looks for a bean named customerPreferenceDao, then falls back to a primary type match for the type CustomerPreferenceDao. The "context" field is injected based on the known resolvable dependency type ApplicationContext.

**public** **class** MovieRecommender {

*@Resource*

**private** CustomerPreferenceDao customerPreferenceDao;

*@Resource*

**private** ApplicationContext context;

**public** MovieRecommender() {

}

*// ...*

}

### 5.9.6 @PostConstruct and @PreDestroy

The CommonAnnotationBeanPostProcessor not only recognizes the @Resource annotation but also the JSR-250 lifecycle annotations. Introduced in Spring 2.5, the support for these annotations offers yet another alternative to those described in [initialization callbacks](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-lifecycle-initializingbean) and [destruction callbacks](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-lifecycle-disposablebean). Provided that the CommonAnnotationBeanPostProcessor is registered within the Spring ApplicationContext, a method carrying one of these annotations is invoked at the same point in the lifecycle as the corresponding Spring lifecycle interface method or explicitly declared callback method. In the example below, the cache will be pre-populated upon initialization and cleared upon destruction.

**public** **class** CachingMovieLister {

*@PostConstruct*

**public** **void** populateMovieCache() {

*// populates the movie cache upon initialization...*

}

*@PreDestroy*

**public** **void** clearMovieCache() {

*// clears the movie cache upon destruction...*

}

}

|  |
| --- |
| [Note] |
| For details about the effects of combining various lifecycle mechanisms, see [the section called “Combining lifecycle mechanisms”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-lifecycle-combined-effects). |

## 5.10 Classpath scanning and managed components

Most examples in this chapter use XML to specify the configuration metadata that produces each BeanDefinition within the Spring container. The previous section ([Section 5.9, “Annotation-based container configuration”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-annotation-config)) demonstrates how to provide a lot of the configuration metadata through source-level annotations. Even in those examples, however, the "base" bean definitions are explicitly defined in the XML file, while the annotations only drive the dependency injection. This section describes an option for implicitly detecting the candidate components by scanning the classpath. Candidate components are classes that match against a filter criteria and have a corresponding bean definition registered with the container. This removes the need to use XML to perform bean registration, instead you can use annotations (for example @Component), AspectJ type expressions, or your own custom filter criteria to select which classes will have bean definitions registered with the container.

|  |
| --- |
| [Note] |
| Starting with Spring 3.0, many features provided by the [Spring JavaConfig project](http://www.springsource.org/javaconfig) are part of the core Spring Framework. This allows you to define beans using Java rather than using the traditional XML files. Take a look at the @Configuration, @Bean, @Import, and @DependsOn annotations for examples of how to use these new features. |

### 5.10.1 @Component and further stereotype annotations

In Spring 2.0 and later, the @Repository annotation is a marker for any class that fulfills the role or stereotype (also known as Data Access Object or DAO) of a repository. Among the uses of this marker is the automatic translation of exceptions as described in [Section 15.2.2, “Exception translation”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/orm.html#orm-exception-translation).

Spring 2.5 introduces further stereotype annotations: @Component, @Service, and @Controller. @Component is a generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively. Therefore, you can annotate your component classes with @Component, but by annotating them with @Repository, @Service, or @Controller instead, your classes are more properly suited for processing by tools or associating with aspects. For example, these stereotype annotations make ideal targets for pointcuts. It is also possible that @Repository, @Service, and @Controller may carry additional semantics in future releases of the Spring Framework. Thus, if you are choosing between using @Component or @Service for your service layer, @Service is clearly the better choice. Similarly, as stated above, @Repository is already supported as a marker for automatic exception translation in your persistence layer.

### 5.10.2 Automatically detecting classes and registering bean definitions

Spring can automatically detect stereotyped classes and register corresponding BeanDefinitions with the ApplicationContext. For example, the following two classes are eligible for such autodetection:

*@Service*

**public** **class** SimpleMovieLister {

**private** MovieFinder movieFinder;

*@Autowired*

**public** SimpleMovieLister(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

}

*@Repository*

**public** **class** JpaMovieFinder **implements** MovieFinder {

*// implementation elided for clarity*

}

To autodetect these classes and register the corresponding beans, you need to include the following element in XML, where the base-package element is a common parent package for the two classes. (Alternatively, you can specify a comma-separated list that includes the parent package of each class.)

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

<context:component-scan base-package="org.example"/>

</beans>

|  |
| --- |
| [Tip] |
| The use of <context:component-scan> implicitly enables the functionality of <context:annotation-config>. There is usually no need to include the <context:annotation-config> element when using <context:component-scan>. |
| [Note] |
| The scanning of classpath packages requires the presence of corresponding directory entries in the classpath. When you build JARs with Ant, make sure that you do not activate the files-only switch of the JAR task. | |

Furthermore, the AutowiredAnnotationBeanPostProcessor and CommonAnnotationBeanPostProcessor are both included implicitly when you use the component-scan element. That means that the two components are autodetected and wired together - all without any bean configuration metadata provided in XML.

|  |
| --- |
| [Note] |
| You can disable the registration of AutowiredAnnotationBeanPostProcessor and CommonAnnotationBeanPostProcessor by including the annotation-config attribute with a value of false. |

### 5.10.3 Using filters to customize scanning

By default, classes annotated with @Component, @Repository, @Service, @Controller, or a custom annotation that itself is annotated with @Component are the only detected candidate components. However, you can modify and extend this behavior simply by applying custom filters. Add them as include-filter or exclude-filter sub-elements of the component-scan element. Each filter element requires the type and expression attributes. The following table describes the filtering options.

**Table 5.5. Filter Types**

| **Filter Type** | **Example Expression** | **Description** |
| --- | --- | --- |
| annotation | org.example.SomeAnnotation | An annotation to be present at the type level in target components. |
| assignable | org.example.SomeClass | A class (or interface) that the target components are assignable to (extend/implement). |
| aspectj | org.example..\*Service+ | An AspectJ type expression to be matched by the target components. |
| regex | org\.example\.Default.\* | A regex expression to be matched by the target components class names. |
| custom | org.example.MyTypeFilter | A custom implementation of the org.springframework.core.type .TypeFilter interface. |

The following example shows the XML configuration ignoring all @Repository annotations and using "stub" repositories instead.

<beans>

<context:component-scan base-package="org.example">

<context:include-filter type="regex" expression=".\*Stub.\*Repository"/>

<context:exclude-filter type="annotation"

expression="org.springframework.stereotype.Repository"/>

</context:component-scan>

</beans>

|  |
| --- |
| [Note] |
| You can also disable the default filters by providing use-default-filters="false" as an attribute of the <component-scan/> element. This will in effect disable automatic detection of classes annotated with @Component, @Repository, @Service, or @Controller. |

### 5.10.4 Defining bean metadata within components

Spring components can also contribute bean definition metadata to the container. You do this with the same @Bean annotation used to define bean metadata within @Configuration annotated classes. Here is a simple example:

*@Component*

**public** **class** FactoryMethodComponent {

*@Bean* *@Qualifier("public")*

**public** TestBean publicInstance() {

**return** **new** TestBean("publicInstance");

}

**public** **void** doWork() {

*// Component method implementation omitted*

}

}

This class is a Spring component that has application-specific code contained in its doWork() method. However, it also contributes a bean definition that has a factory method referring to the method publicInstance(). The @Bean annotation identifies the factory method and other bean definition properties, such as a qualifier value through the @Qualifier annotation. Other method level annotations that can be specified are @Scope, @Lazy, and custom qualifier annotations. Autowired fields and methods are supported as previously discussed, with additional support for autowiring of @Bean methods:

*@Component*

**public** **class** FactoryMethodComponent {

**private** **static** **int** i;

*@Bean* *@Qualifier("public")*

**public** TestBean publicInstance() {

**return** **new** TestBean("publicInstance");

}

*// use of a custom qualifier and autowiring of method parameters*

*@Bean*

**protected** TestBean protectedInstance(*@Qualifier("public")* TestBean spouse,

*@Value("#{privateInstance.age}")* String country) {

TestBean tb = **new** TestBean("protectedInstance", 1);

tb.setSpouse(tb);

tb.setCountry(country);

**return** tb;

}

*@Bean* *@Scope(BeanDefinition.SCOPE\_SINGLETON)*

**private** TestBean privateInstance() {

**return** **new** TestBean("privateInstance", i++);

}

*@Bean* *@Scope(value = WebApplicationContext.SCOPE\_SESSION,*

*proxyMode = ScopedProxyMode.TARGET\_CLASS)*

**public** TestBean requestScopedInstance() {

**return** **new** TestBean("requestScopedInstance", 3);

}

}

The example autowires the String method parameter country to the value of the Age property on another bean named privateInstance. A Spring Expression Language element defines the value of the property through the notation #{ <expression> }. For @Value annotations, an expression resolver is preconfigured to look for bean names when resolving expression text.

The @Bean methods in a Spring component are processed differently than their counterparts inside a Spring @Configuration class. The difference is that @Component classes are not enhanced with CGLIB to intercept the invocation of methods and fields. CGLIB proxying is the means by which invoking methods or fields within @Configuration classes @Bean methods create bean metadata references to collaborating objects. Methods are not invoked with normal Java semantics. In contrast, calling a method or field within a @Component classes @Bean method has standard Java semantics.

### 5.10.5 Naming autodetected components

When a component is autodetected as part of the scanning process, its bean name is generated by the BeanNameGenerator strategy known to that scanner. By default, any Spring stereotype annotation (@Component, @Repository, @Service, and @Controller) that contains a name value will thereby provide that name to the corresponding bean definition.

If such an annotation contains no name value or for any other detected component (such as those discovered by custom filters), the default bean name generator returns the uncapitalized non-qualified class name. For example, if the following two components were detected, the names would be myMovieLister and movieFinderImpl:

*@Service("myMovieLister")*

**public** **class** SimpleMovieLister {

*// ...*

}

*@Repository*

**public** **class** MovieFinderImpl **implements** MovieFinder {

*// ...*

}

|  |
| --- |
| [Note] |
| If you do not want to rely on the default bean-naming strategy, you can provide a custom bean-naming strategy. First, implement the [BeanNameGenerator](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/support/BeanNameGenerator.html) interface, and be sure to include a default no-arg constructor. Then, provide the fully-qualified class name when configuring the scanner: |

<beans>

<context:component-scan base-package="org.example"

name-generator="org.example.MyNameGenerator" />

</beans>

As a general rule, consider specifying the name with the annotation whenever other components may be making explicit references to it. On the other hand, the auto-generated names are adequate whenever the container is responsible for wiring.

### 5.10.6 Providing a scope for autodetected components

As with Spring-managed components in general, the default and most common scope for autodetected components is singleton. However, sometimes you need other scopes, which Spring 2.5 provides with a new @Scope annotation. Simply provide the name of the scope within the annotation:

*@Scope("prototype")*

*@Repository*

**public** **class** MovieFinderImpl **implements** MovieFinder {

*// ...*

}

|  |
| --- |
| [Note] |
| To provide a custom strategy for scope resolution rather than relying on the annotation-based approach, implement the [ScopeMetadataResolver](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/ScopeMetadataResolver.html) interface, and be sure to include a default no-arg constructor. Then, provide the fully-qualified class name when configuring the scanner: |

<beans>

<context:component-scan base-package="org.example"

scope-resolver="org.example.MyScopeResolver" />

</beans>

When using certain non-singleton scopes, it may be necessary to generate proxies for the scoped objects. The reasoning is described in [the section called “Scoped beans as dependencies”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes-other-injection). For this purpose, a scoped-proxy attribute is available on the component-scan element. The three possible values are: no, interfaces, and targetClass. For example, the following configuration will result in standard JDK dynamic proxies:

<beans>

<context:component-scan base-package="org.example"

scoped-proxy="interfaces" />

</beans>

### 5.10.7 Providing qualifier metadata with annotations

The @Qualifier annotation is discussed in [Section 5.9.3, “Fine-tuning annotation-based autowiring with qualifiers”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-autowired-annotation-qualifiers). The examples in that section demonstrate the use of the @Qualifier annotation and custom qualifier annotations to provide fine-grained control when you resolve autowire candidates. Because those examples were based on XML bean definitions, the qualifier metadata was provided on the candidate bean definitions using the qualifier or meta sub-elements of the bean element in the XML. When relying upon classpath scanning for autodetection of components, you provide the qualifier metadata with type-level annotations on the candidate class. The following three examples demonstrate this technique:

*@Component*

**@Qualifier("Action")**

**public** **class** ActionMovieCatalog **implements** MovieCatalog {

*// ...*

}

*@Component*

**@Genre("Action")**

**public** **class** ActionMovieCatalog **implements** MovieCatalog {

*// ...*

}

*@Component*

**@Offline**

**public** **class** CachingMovieCatalog **implements** MovieCatalog {

*// ...*

}

|  |
| --- |
| [Note] |
| As with most annotation-based alternatives, keep in mind that the annotation metadata is bound to the class definition itself, while the use of XML allows for multiple beans of the same type to provide variations in their qualifier metadata, because that metadata is provided per-instance rather than per-class. |

## 5.11 Using JSR 330 Standard Annotations

Starting with Spring 3.0, Spring offers support for JSR-330 standard annotations (Dependency Injection). Those annotations are scanned in the same way as the Spring annotations. You just need to have the relevant jars in your classpath.

|  |
| --- |
| [Note] |
| If you are using Maven, the javax.inject artifact is available in the standard Maven repository ([http://repo1.maven.org/maven2/javax/inject/javax.inject/1/](https://repo1.maven.org/maven2/javax/inject/javax.inject/1/)). You can add the following dependency to your file pom.xml:  <dependency>  <groupId>javax.inject</groupId>  <artifactId>javax.inject</artifactId>  <version>1</version>  </dependency> |

### 5.11.1 Dependency Injection with @Inject and @Named

Instead of @Autowired, @javax.inject.Inject may be used as follows:

**import** javax.inject.Inject;

**public** **class** SimpleMovieLister {

**private** MovieFinder movieFinder;

*@Inject*

**public** **void** setMovieFinder(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

*// ...*

}

As with @Autowired, it is possible to use @Inject at the class-level, field-level, method-level and constructor-argument level. If you would like to use a qualified name for the dependency that should be injected, you should use the @Named annotation as follows:

**import** javax.inject.Inject;

**import** javax.inject.Named;

**public** **class** SimpleMovieLister {

**private** MovieFinder movieFinder;

*@Inject*

**public** **void** setMovieFinder(*@Named("main")* MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

*// ...*

}

### 5.11.2 @Named: a standard equivalent to the @Component annotation

Instead of @Component, @javax.inject.Named may be used as follows:

**import** javax.inject.Inject;

**import** javax.inject.Named;

*@Named("movieListener")*

**public** **class** SimpleMovieLister {

**private** MovieFinder movieFinder;

*@Inject*

**public** **void** setMovieFinder(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

*// ...*

}

It is very common to use @Component without specifying a name for the component. @Named can be used in a similar fashion:

**import** javax.inject.Inject;

**import** javax.inject.Named;

*@Named*

**public** **class** SimpleMovieLister {

**private** MovieFinder movieFinder;

*@Inject*

**public** **void** setMovieFinder(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

*// ...*

}

When using @Named, it is possible to use component-scanning in the exact same way as when using Spring annotations:

<beans>

<context:component-scan base-package="org.example"/>

</beans>

### 5.11.3 Limitations of the standard approach

When working with standard annotations, it is important to know that some significant features are not available as shown in the table below:

**Table 5.6. Spring annotations vs. standard annotations**

| **Spring** | **javax.inject.\*** | **javax.inject restrictions / comments** |
| --- | --- | --- |
| @Autowired | @Inject | @Inject has no 'required' attribute |
| @Component | @Named | — |
| @Scope("singleton") | @Singleton | The JSR-330 default scope is like Spring's prototype. However, in order to keep it consistent with Spring's general defaults, a JSR-330 bean declared in the Spring container is a singleton by default. In order to use a scope other than singleton, you should use Spring's @Scope annotation.  javax.inject also provides a [@Scope](https://download.oracle.com/javaee/6/api/javax/inject/Scope.html) annotation. Nevertheless, this one is only intended to be used for creating your own annotations. |
| @Qualifier | @Named | — |
| @Value | — | no equivalent |
| @Required | — | no equivalent |
| @Lazy | — | no equivalent |

## 5.12 Java-based container configuration

### 5.12.1 Basic concepts: @Bean and @Configuration

**Full @Configuration vs 'lite' @Beans mode?**

When @Bean methods are declared within classes that are not annotated with @Configuration they are referred to as being processed in a 'lite' mode. For example, bean methods declared in a @Component or even in a plain old class will be considered 'lite'.

Unlike full @Configuration, lite @Bean methods cannot easily declare inter-bean dependencies. Usually one @Bean method should not invoke another @Bean method when operating in 'lite' mode.

Only using @Bean methods within @Configuration classes is a recommended approach of ensuring that 'full' mode is always used. This will prevent the same @Bean method from accidentally being invoked multiple times and helps to reduce subtle bugs that can be hard to track down when operating in 'lite' mode.

The central artifacts in Spring's new Java-configuration support are @Configuration-annotated classes and @Bean-annotated methods.

The @Bean annotation is used to indicate that a method instantiates, configures and initializes a new object to be managed by the Spring IoC container. For those familiar with Spring's <beans/> XML configuration the @Bean annotation plays the same role as the <bean/> element. You can use @Bean annotated methods with any Spring @Component, however, they are most often used with @Configuration beans.

Annotating a class with @Configuration indicates that its primary purpose is as a source of bean definitions. Furthermore, @Configuration classes allow inter-bean dependencies to be defined by simply calling other @Bean methods in the same class. The simplest possible @Configuration class would read as follows:

*@Configuration*

**public** **class** AppConfig {

*@Bean*

**public** MyService myService() {

**return** **new** MyServiceImpl();

}

}

The AppConfig class above would be equivalent to the following Spring <beans/> XML:

<beans>

<bean id="myService" class="com.acme.services.MyServiceImpl"/>

</beans>

The @Bean and @Configuration annotations will be discussed in depth in the sections below. First, however, we'll cover the various ways of creating a spring container using Java-based configuration.

### 5.12.2 Instantiating the Spring container using AnnotationConfigApplicationContext

The sections below document Spring's AnnotationConfigApplicationContext, new in Spring 3.0. This versatile ApplicationContext implementation is capable of accepting not only @Configuration classes as input, but also plain @Component classes and classes annotated with JSR-330 metadata.

When @Configuration classes are provided as input, the @Configuration class itself is registered as a bean definition, and all declared @Bean methods within the class are also registered as bean definitions.

When @Component and JSR-330 classes are provided, they are registered as bean definitions, and it is assumed that DI metadata such as @Autowired or @Inject are used within those classes where necessary.

#### Simple construction

In much the same way that Spring XML files are used as input when instantiating a ClassPathXmlApplicationContext, @Configuration classes may be used as input when instantiating an AnnotationConfigApplicationContext. This allows for completely XML-free usage of the Spring container:

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(AppConfig.**class**);

MyService myService = ctx.getBean(MyService.**class**);

myService.doStuff();

}

As mentioned above, AnnotationConfigApplicationContext is not limited to working only with @Configuration classes. Any @Component or JSR-330 annotated class may be supplied as input to the constructor. For example:

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(MyServiceImpl.**class**, Dependency1.**class**, Dependency2.**class**);

MyService myService = ctx.getBean(MyService.**class**);

myService.doStuff();

}

The above assumes that MyServiceImpl, Dependency1 and Dependency2 use Spring dependency injection annotations such as @Autowired.

#### Building the container programmatically using register(Class<?>...)

An AnnotationConfigApplicationContext may be instantiated using a no-arg constructor and then configured using the register() method. This approach is particularly useful when programmatically building an AnnotationConfigApplicationContext.

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

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.register(AppConfig.**class**, OtherConfig.**class**);

ctx.register(AdditionalConfig.**class**);

ctx.refresh();

MyService myService = ctx.getBean(MyService.**class**);

myService.doStuff();

}

#### Enabling component scanning with scan(String...)

Experienced Spring users will be familiar with the following commonly-used XML declaration from Spring's context: namespace

<beans>

<context:component-scan base-package="com.acme"/>

</beans>

In the example above, the com.acme package will be scanned, looking for any @Component-annotated classes, and those classes will be registered as Spring bean definitions within the container. AnnotationConfigApplicationContext exposes the scan(String...) method to allow for the same component-scanning functionality:

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

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.scan("com.acme");

ctx.refresh();

MyService myService = ctx.getBean(MyService.**class**);

}

|  |
| --- |
| [Note] |
| Remember that @Configuration classes are meta-annotated with @Component, so they are candidates for component-scanning! In the example above, assuming that AppConfig is declared within the com.acme package (or any package underneath), it will be picked up during the call to scan(), and upon refresh() all its @Bean methods will be processed and registered as bean definitions within the container. |

#### Support for web applications with AnnotationConfigWebApplicationContext

A WebApplicationContext variant of AnnotationConfigApplicationContext is available with AnnotationConfigWebApplicationContext. This implementation may be used when configuring the Spring ContextLoaderListener servlet listener, Spring MVC DispatcherServlet, etc. What follows is a web.xml snippet that configures a typical Spring MVC web application. Note the use of the contextClass context-param and init-param:

<web-app>

*<!-- Configure ContextLoaderListener to use AnnotationConfigWebApplicationContext*

*instead of the default XmlWebApplicationContext -->*

<context-param>

<param-name>contextClass</param-name>

<param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext

</param-value>

</context-param>

*<!-- Configuration locations must consist of one or more comma- or space-delimited*

*fully-qualified @Configuration classes. Fully-qualified packages may also be*

*specified for component-scanning -->*

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>com.acme.AppConfig</param-value>

</context-param>

*<!-- Bootstrap the root application context as usual using ContextLoaderListener -->*

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

*<!-- Declare a Spring MVC DispatcherServlet as usual -->*

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

*<!-- Configure DispatcherServlet to use AnnotationConfigWebApplicationContext*

*instead of the default XmlWebApplicationContext -->*

<init-param>

<param-name>contextClass</param-name>

<param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext

</param-value>

</init-param>

*<!-- Again, config locations must consist of one or more comma- or space-delimited*

*and fully-qualified @Configuration classes -->*

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>com.acme.web.MvcConfig</param-value>

</init-param>

</servlet>

*<!-- map all requests for /app/\* to the dispatcher servlet -->*

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/app/\*</url-pattern>

</servlet-mapping>

</web-app>

### 5.12.3 Using the @Bean annotation

@Bean is a method-level annotation and a direct analog of the XML <bean/> element. The annotation supports some of the attributes offered by <bean/>, such as: [init-method](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-lifecycle-initializingbean), [destroy-method](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-lifecycle-disposablebean), [autowiring](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-autowire) and name.

You can use the @Bean annotation in a @Configuration-annotated or in a @Component-annotated class.

#### Declaring a bean

To declare a bean, simply annotate a method with the @Bean annotation. You use this method to register a bean definition within an ApplicationContext of the type specified as the method's return value. By default, the bean name will be the same as the method name. The following is a simple example of a @Bean method declaration:

*@Configuration*

**public** **class** AppConfig {

*@Bean*

**public** TransferService transferService() {

**return** **new** TransferServiceImpl();

}

}

The preceding configuration is exactly equivalent to the following Spring XML:

<beans>

<bean id="transferService" class="com.acme.TransferServiceImpl"/>

</beans>

Both declarations make a bean named transferService available in the ApplicationContext, bound to an object instance of type TransferServiceImpl:

transferService -> com.acme.TransferServiceImpl

#### Receiving lifecycle callbacks

Any classes defined with the @Bean annotation support the regular lifecycle callbacks and can use the @PostConstruct and @PreDestroy annotations from JSR-250, see [JSR-250 annotations](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-postconstruct-and-predestroy-annotations) for further details.

The regular Spring [lifecycle](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-nature) callbacks are fully supported as well. If a bean implements InitializingBean, DisposableBean, or Lifecycle, their respective methods are called by the container.

The standard set of \*Aware interfaces such as [BeanFactoryAware](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-beanfactory), [BeanNameAware](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-aware), [MessageSourceAware](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#context-functionality-messagesource), [ApplicationContextAware](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-aware), and so on are also fully supported.

The @Bean annotation supports specifying arbitrary initialization and destruction callback methods, much like Spring XML's init-method and destroy-method attributes on the bean element:

**public** **class** Foo {

**public** **void** init() {

*// initialization logic*

}

}

**public** **class** Bar {

**public** **void** cleanup() {

*// destruction logic*

}

}

*@Configuration*

**public** **class** AppConfig {

*@Bean(initMethod = "init")*

**public** Foo foo() {

**return** **new** Foo();

}

*@Bean(destroyMethod = "cleanup")*

**public** Bar bar() {

**return** **new** Bar();

}

}

Of course, in the case of Foo above, it would be equally as valid to call the init() method directly during construction:

*@Configuration*

**public** **class** AppConfig {

*@Bean*

**public** Foo foo() {

Foo foo = **new** Foo();

foo.init();

**return** foo;

}

*// ...*

}

|  |
| --- |
| [Tip] |
| When you work directly in Java, you can do anything you like with your objects and do not always need to rely on the container lifecycle! |

#### Specifying bean scope

##### Using the @Scope annotation

You can specify that your beans defined with the @Bean annotation should have a specific scope. You can use any of the standard scopes specified in the [Bean Scopes](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes) section.

The default scope is singleton, but you can override this with the @Scope annotation:

*@Configuration*

**public** **class** MyConfiguration {

*@Bean*

**@Scope("prototype")**

**public** Encryptor encryptor() {

*// ...*

}

}

##### @Scope and scoped-proxy

Spring offers a convenient way of working with scoped dependencies through [scoped proxies](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-scopes-other-injection). The easiest way to create such a proxy when using the XML configuration is the <aop:scoped-proxy/> element. Configuring your beans in Java with a @Scope annotation offers equivalent support with the proxyMode attribute. The default is no proxy (ScopedProxyMode.NO), but you can specify ScopedProxyMode.TARGET\_CLASS or ScopedProxyMode.INTERFACES.

If you port the scoped proxy example from the XML reference documentation (see preceding link) to our @Bean using Java, it would look like the following:

*// an HTTP Session-scoped bean exposed as a proxy*

*@Bean*

**@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET\_CLASS)**

**public** UserPreferences userPreferences() {

**return** **new** UserPreferences();

}

*@Bean*

**public** Service userService() {

UserService service = **new** SimpleUserService();

*// a reference to the proxied userPreferences bean*

service.setUserPreferences(userPreferences());

**return** service;

}

#### Customizing bean naming

By default, configuration classes use a @Bean method's name as the name of the resulting bean. This functionality can be overridden, however, with the name attribute.

*@Configuration*

**public** **class** AppConfig {

*@Bean(name = "myFoo")*

**public** Foo foo() {

**return** **new** Foo();

}

}

#### Bean aliasing

As discussed in [Section 5.3.1, “Naming beans”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-beanname), it is sometimes desirable to give a single bean multiple names, otherwise known as bean aliasing. The name attribute of the @Bean annotation accepts a String array for this purpose.

*@Configuration*

**public** **class** AppConfig {

*@Bean(name = { "dataSource", "subsystemA-dataSource", "subsystemB-dataSource" })*

**public** DataSource dataSource() {

*// instantiate, configure and return DataSource bean...*

}

}

### 5.12.4 Using the @Configuration annotation

@Configuration is a class-level annotation indicating that an object is a source of bean definitions. @Configuration classes declare beans via public @Bean annotated methods. Calls to @Bean methods on @Configuration classes can also be used to define inter-bean dependencies. See [Section 5.12.1, “Basic concepts: @Bean and @Configuration”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-java-basic-concepts) for a general introduction.

#### Injecting inter-bean dependencies

When @Beans have dependencies on one another, expressing that dependency is as simple as having one bean method call another:

*@Configuration*

**public** **class** AppConfig {

*@Bean*

**public** Foo foo() {

**return** **new** Foo(bar());

}

*@Bean*

**public** Bar bar() {

**return** **new** Bar();

}

}

In the example above, the foo bean receives a reference to bar via constructor injection.

|  |
| --- |
| [Note] |
| This method of declaring inter-bean dependencies only works when the @Bean method is declared within a @Configuration class. You cannot declare inter-bean dependencies using plain @Component classes. |

#### Lookup method injection

As noted earlier, [lookup method injection](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-method-injection) is an advanced feature that you should use rarely. It is useful in cases where a singleton-scoped bean has a dependency on a prototype-scoped bean. Using Java for this type of configuration provides a natural means for implementing this pattern.

**public** **abstract** **class** CommandManager {

**public** Object process(Object commandState) {

*// grab a new instance of the appropriate Command interface*

Command command = createCommand();

*// set the state on the (hopefully brand new) Command instance*

command.setState(commandState);

**return** command.execute();

}

*// okay... but where is the implementation of this method?*

**protected** **abstract** Command createCommand();

}

Using Java-configuration support , you can create a subclass of CommandManager where the abstract createCommand() method is overridden in such a way that it looks up a new (prototype) command object:

*@Bean*

*@Scope("prototype")*

**public** AsyncCommand asyncCommand() {

AsyncCommand command = **new** AsyncCommand();

*// inject dependencies here as required*

**return** command;

}

*@Bean*

**public** CommandManager commandManager() {

*// return new anonymous implementation of CommandManager with command() overridden*

*// to return a new prototype Command object*

**return** **new** CommandManager() {

**protected** Command createCommand() {

**return** asyncCommand();

}

}

}

#### Further information about how Java-based configuration works internally

The following example shows a @Bean annotated method being called twice:

*@Configuration*

**public** **class** AppConfig {

*@Bean*

**public** ClientService clientService1() {

ClientServiceImpl clientService = **new** ClientServiceImpl();

clientService.setClientDao(clientDao());

**return** clientService;

}

*@Bean*

**public** ClientService clientService2() {

ClientServiceImpl clientService = **new** ClientServiceImpl();

clientService.setClientDao(clientDao());

**return** clientService;

}

*@Bean*

**public** ClientDao clientDao() {

**return** **new** ClientDaoImpl();

}

}

clientDao() has been called once in clientService1() and once in clientService2(). Since this method creates a new instance of ClientDaoImpl and returns it, you would normally expect having 2 instances (one for each service). That definitely would be problematic: in Spring, instantiated beans have a singleton scope by default. This is where the magic comes in: All @Configuration classes are subclassed at startup-time with CGLIB. In the subclass, the child method checks the container first for any cached (scoped) beans before it calls the parent method and creates a new instance. Note that as of Spring 3.2, it is no longer necessary to add CGLIB to your classpath because CGLIB classes have been repackaged under org.springframework and included directly within the spring-core JAR.

|  |
| --- |
| [Note] |
| The behavior could be different according to the scope of your bean. We are talking about singletons here. |
| [Note] |
| There are a few restrictions due to the fact that CGLIB dynamically adds features at startup-time:   * Configuration classes should not be final * They should have a constructor with no arguments | |

### 5.12.5 Composing Java-based configurations

#### Using the @Import annotation

Much as the <import/> element is used within Spring XML files to aid in modularizing configurations, the @Import annotation allows for loading @Bean definitions from another configuration class:

*@Configuration*

**public** **class** ConfigA {

**public** *@Bean* A a() { **return** **new** A(); }

}

*@Configuration*

*@Import(ConfigA.class)*

**public** **class** ConfigB {

**public** *@Bean* B b() { **return** **new** B(); }

}

Now, rather than needing to specify both ConfigA.class and ConfigB.class when instantiating the context, only ConfigB needs to be supplied explicitly:

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(ConfigB.**class**);

*// now both beans A and B will be available...*

A a = ctx.getBean(A.**class**);

B b = ctx.getBean(B.**class**);

}

This approach simplifies container instantiation, as only one class needs to be dealt with, rather than requiring the developer to remember a potentially large number of @Configuration classes during construction.

##### Injecting dependencies on imported @Bean definitions

The example above works, but is simplistic. In most practical scenarios, beans will have dependencies on one another across configuration classes. When using XML, this is not an issue, per se, because there is no compiler involved, and one can simply declare ref="someBean" and trust that Spring will work it out during container initialization. Of course, when using @Configuration classes, the Java compiler places constraints on the configuration model, in that references to other beans must be valid Java syntax.

Fortunately, solving this problem is simple. Remember that @Configuration classes are ultimately just another bean in the container - this means that they can take advantage of @Autowired injection metadata just like any other bean!

Let's consider a more real-world scenario with several @Configuration classes, each depending on beans declared in the others:

*@Configuration*

**public** **class** ServiceConfig {

**private** *@Autowired* AccountRepository accountRepository;

**public** *@Bean* TransferService transferService() {

**return** **new** TransferServiceImpl(accountRepository);

}

}

*@Configuration*

**public** **class** RepositoryConfig {

**private** *@Autowired* DataSource dataSource;

**public** *@Bean* AccountRepository accountRepository() {

**return** **new** JdbcAccountRepository(dataSource);

}

}

*@Configuration*

*@Import({ServiceConfig.class, RepositoryConfig.class})*

**public** **class** SystemTestConfig {

**public** *@Bean* DataSource dataSource() { */\* return new DataSource \*/* }

}

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(SystemTestConfig.**class**);

*// everything wires up across configuration classes...*

TransferService transferService = ctx.getBean(TransferService.**class**);

transferService.transfer(100.00, "A123", "C456");

}

###### Fully-qualifying imported beans for ease of navigation

In the scenario above, using @Autowired works well and provides the desired modularity, but determining exactly where the autowired bean definitions are declared is still somewhat ambiguous. For example, as a developer looking at ServiceConfig, how do you know exactly where the @Autowired AccountRepository bean is declared? It's not explicit in the code, and this may be just fine. Remember that the [SpringSource Tool Suite](http://www.springsource.com/products/sts) provides tooling that can render graphs showing how everything is wired up - that may be all you need. Also, your Java IDE can easily find all declarations and uses of the AccountRepository type, and will quickly show you the location of @Bean methods that return that type.

In cases where this ambiguity is not acceptable and you wish to have direct navigation from within your IDE from one @Configuration class to another, consider autowiring the configuration classes themselves:

*@Configuration*

**public** **class** ServiceConfig {

**private** *@Autowired* RepositoryConfig repositoryConfig;

**public** *@Bean* TransferService transferService() {

*// navigate 'through' the config class to the @Bean method!*

**return** **new** TransferServiceImpl(repositoryConfig.accountRepository());

}

}

In the situation above, it is completely explicit where AccountRepository is defined. However, ServiceConfig is now tightly coupled to RepositoryConfig; that's the tradeoff. This tight coupling can be somewhat mitigated by using interface-based or abstract class-based @Configuration classes. Consider the following:

*@Configuration*

**public** **class** ServiceConfig {

**private** *@Autowired* RepositoryConfig repositoryConfig;

**public** *@Bean* TransferService transferService() {

**return** **new** TransferServiceImpl(repositoryConfig.accountRepository());

}

}

*@Configuration*

**public** **interface** RepositoryConfig {

*@Bean* AccountRepository accountRepository();

}

*@Configuration*

**public** **class** DefaultRepositoryConfig **implements** RepositoryConfig {

**public** *@Bean* AccountRepository accountRepository() {

**return** **new** JdbcAccountRepository(...);

}

}

*@Configuration*

*@Import({ServiceConfig.class, DefaultRepositoryConfig.class})* *// import the concrete config!*

**public** **class** SystemTestConfig {

**public** *@Bean* DataSource dataSource() { */\* return DataSource \*/* }

}

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(SystemTestConfig.**class**);

TransferService transferService = ctx.getBean(TransferService.**class**);

transferService.transfer(100.00, "A123", "C456");

}

Now ServiceConfig is loosely coupled with respect to the concrete DefaultRepositoryConfig, and built-in IDE tooling is still useful: it will be easy for the developer to get a type hierarchy of RepositoryConfig implementations. In this way, navigating @Configuration classes and their dependencies becomes no different than the usual process of navigating interface-based code.

#### Combining Java and XML configuration

Spring's @Configuration class support does not aim to be a 100% complete replacement for Spring XML. Some facilities such as Spring XML namespaces remain an ideal way to configure the container. In cases where XML is convenient or necessary, you have a choice: either instantiate the container in an "XML-centric" way using, for example, ClassPathXmlApplicationContext, or in a "Java-centric" fashion using AnnotationConfigApplicationContext and the @ImportResource annotation to import XML as needed.

##### XML-centric use of @Configuration classes

It may be preferable to bootstrap the Spring container from XML and include @Configuration classes in an ad-hoc fashion. For example, in a large existing codebase that uses Spring XML, it will be easier to create @Configuration classes on an as-needed basis and include them from the existing XML files. Below you'll find the options for using @Configuration classes in this kind of "XML-centric" situation.

###### Declaring @Configuration classes as plain Spring <bean/> elements

Remember that @Configuration classes are ultimately just bean definitions in the container. In this example, we create a @Configuration class named AppConfig and include it within system-test-config.xml as a <bean/>definition. Because <context:annotation-config/> is switched on, the container will recognize the @Configuration annotation, and process the @Bean methods declared in AppConfig properly.

*@Configuration*

**public** **class** AppConfig {

**private** *@Autowired* DataSource dataSource;

**public** *@Bean* AccountRepository accountRepository() {

**return** **new** JdbcAccountRepository(dataSource);

}

**public** *@Bean* TransferService transferService() {

**return** **new** TransferService(accountRepository());

}

}

*system-test-config.xml*

<beans>

*<!-- enable processing of annotations such as @Autowired and @Configuration -->*

<context:annotation-config/>

<context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

<bean class="com.acme.AppConfig"/>

<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

</beans>

*jdbc.properties*

jdbc.url=jdbc:hsqldb:hsql://localhost/xdb

jdbc.username=sa

jdbc.password=

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

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("classpath:/com/acme/system-test-config.xml");

TransferService transferService = ctx.getBean(TransferService.**class**);

*// ...*

}

|  |
| --- |
| [Note] |
| In system-test-config.xml above, the AppConfig<bean/> does not declare an id element. While it would be acceptable to do so, it is unnecessary given that no other bean will ever refer to it, and it is unlikely that it will be explicitly fetched from the container by name. Likewise with the DataSource bean - it is only ever autowired by type, so an explicit bean id is not strictly required. |

###### Using <context:component-scan/> to pick up @Configuration classes

Because @Configuration is meta-annotated with @Component, @Configuration-annotated classes are automatically candidates for component scanning. Using the same scenario as above, we can redefine system-test-config.xml to take advantage of component-scanning. Note that in this case, we don't need to explicitly declare <context:annotation-config/>, because <context:component-scan/> enables all the same functionality.

*system-test-config.xml*

<beans>

*<!-- picks up and registers AppConfig as a bean definition -->*

<context:component-scan base-package="com.acme"/>

<context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

</beans>

##### @Configuration class-centric use of XML with @ImportResource

In applications where @Configuration classes are the primary mechanism for configuring the container, it will still likely be necessary to use at least some XML. In these scenarios, simply use @ImportResource and define only as much XML as is needed. Doing so achieves a "Java-centric" approach to configuring the container and keeps XML to a bare minimum.

*@Configuration*

*@ImportResource("classpath:/com/acme/properties-config.xml")*

**public** **class** AppConfig {

**private** *@Value("${jdbc.url}")* String url;

**private** *@Value("${jdbc.username}")* String username;

**private** *@Value("${jdbc.password}")* String password;

**public** *@Bean* DataSource dataSource() {

**return** **new** DriverManagerDataSource(url, username, password);

}

}

*properties-config.xml*

<beans>

<context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

</beans>

*jdbc.properties*

jdbc.url=jdbc:hsqldb:hsql://localhost/xdb

jdbc.username=sa

jdbc.password=

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(AppConfig.**class**);

TransferService transferService = ctx.getBean(TransferService.**class**);

*// ...*

}

## 5.13 Registering a LoadTimeWeaver

The LoadTimeWeaver is used by Spring to dynamically transform classes as they are loaded into the Java virtual machine (JVM).

To enable load-time weaving add the @EnableLoadTimeWeaving to one of your @Configuration classes:

*@Configuration*

*@EnableLoadTimeWeaving*

**public** **class** AppConfig {

}

Alternatively for XML configuration use the context:load-time-weaver element:

<beans>

<context:load-time-weaver/>

</beans>

Once configured for the ApplicationContext. Any bean within that ApplicationContext may implement LoadTimeWeaverAware, thereby receiving a reference to the load-time weaver instance. This is particularly useful in combination with [Spring's JPA support](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/orm.html#orm-jpa) where load-time weaving may be necessary for JPA class transformation. Consult the LocalContainerEntityManagerFactoryBean Javadoc for more detail. For more on AspectJ load-time weaving, see [Section 9.8.4, “Load-time weaving with AspectJ in the Spring Framework”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/aop.html#aop-aj-ltw).

## 5.14 Additional Capabilities of the ApplicationContext

As was discussed in the chapter introduction, the org.springframework.beans.factory package provides basic functionality for managing and manipulating beans, including in a programmatic way. The org.springframework.context package adds the [ApplicationContext](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/context/ApplicationContext.html) interface, which extends the BeanFactory interface, in addition to extending other interfaces to provide additional functionality in a more application framework-oriented style. Many people use the ApplicationContext in a completely declarative fashion, not even creating it programmatically, but instead relying on support classes such as ContextLoader to automatically instantiate an ApplicationContext as part of the normal startup process of a J2EE web application.

To enhance BeanFactory functionality in a more framework-oriented style the context package also provides the following functionality:

* Access to messages in i18n-style, through the MessageSource interface.
* Access to resources, such as URLs and files, through the ResourceLoader interface.
* Event publication to beans implementing the ApplicationListener interface, through the use of the ApplicationEventPublisher interface.
* Loading of multiple (hierarchical) contexts, allowing each to be focused on one particular layer, such as the web layer of an application, through the HierarchicalBeanFactory interface.

### 5.14.1 Internationalization using MessageSource

The ApplicationContext interface extends an interface called MessageSource, and therefore provides internationalization (i18n) functionality. Spring also provides the interface HierarchicalMessageSource, which can resolve messages hierarchically. Together these interfaces provide the foundation upon which Spring effects message resolution. The methods defined on these interfaces include:

* String getMessage(String code, Object[] args, String default, Locale loc): The basic method used to retrieve a message from the MessageSource. When no message is found for the specified locale, the default message is used. Any arguments passed in become replacement values, using the MessageFormat functionality provided by the standard library.
* String getMessage(String code, Object[] args, Locale loc): Essentially the same as the previous method, but with one difference: no default message can be specified; if the message cannot be found, a NoSuchMessageException is thrown.
* String getMessage(MessageSourceResolvable resolvable, Locale locale): All properties used in the preceding methods are also wrapped in a class named MessageSourceResolvable, which you can use with this method.

When an ApplicationContext is loaded, it automatically searches for a MessageSource bean defined in the context. The bean must have the name messageSource. If such a bean is found, all calls to the preceding methods are delegated to the message source. If no message source is found, the ApplicationContext attempts to find a parent containing a bean with the same name. If it does, it uses that bean as the MessageSource. If the ApplicationContext cannot find any source for messages, an empty DelegatingMessageSource is instantiated in order to be able to accept calls to the methods defined above.

Spring provides two MessageSource implementations, ResourceBundleMessageSource and StaticMessageSource. Both implement HierarchicalMessageSource in order to do nested messaging. The StaticMessageSource is rarely used but provides programmatic ways to add messages to the source. The ResourceBundleMessageSource is shown in the following example:

<beans>

<bean id="messageSource"

class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basenames">

<list>

<value>format</value>

<value>exceptions</value>

<value>windows</value>

</list>

</property>

</bean>

</beans>

In the example it is assumed you have three resource bundles defined in your classpath called format, exceptions and windows. Any request to resolve a message will be handled in the JDK standard way of resolving messages through ResourceBundles. For the purposes of the example, assume the contents of two of the above resource bundle files are...

*# in format.properties*

message=Alligators rock!

*# in exceptions.properties*

argument.required=The {0} argument is required.

A program to execute the MessageSource functionality is shown in the next example. Remember that all ApplicationContext implementations are also MessageSource implementations and so can be cast to the MessageSource interface.

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

MessageSource resources = **new** ClassPathXmlApplicationContext("beans.xml");

String message = resources.getMessage("message", null, "Default", null);

System.out.println(message);

}

The resulting output from the above program will be...

Alligators rock!

So to summarize, the MessageSource is defined in a file called beans.xml, which exists at the root of your classpath. The messageSource bean definition refers to a number of resource bundles through its basenames property. The three files that are passed in the list to the basenames property exist as files at the root of your classpath and are called format.properties, exceptions.properties, and windows.properties respectively.

The next example shows arguments passed to the message lookup; these arguments will be converted into Strings and inserted into placeholders in the lookup message.

<beans>

*<!-- this MessageSource is being used in a web application -->*

<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basename" value="exceptions"/>

</bean>

*<!-- lets inject the above MessageSource into this POJO -->*

<bean id="example" class="com.foo.Example">

<property name="messages" ref="messageSource"/>

</bean>

</beans>

**public** **class** Example {

**private** MessageSource messages;

**public** **void** setMessages(MessageSource messages) {

**this**.messages = messages;

}

**public** **void** execute() {

String message = **this**.messages.getMessage("argument.required",

**new** Object [] {"userDao"}, "Required", null);

System.out.println(message);

}

}

The resulting output from the invocation of the execute() method will be...

The userDao argument is required.

With regard to internationalization (i18n), Spring's various MessageSource implementations follow the same locale resolution and fallback rules as the standard JDK ResourceBundle. In short, and continuing with the example messageSource defined previously, if you want to resolve messages against the British (en-GB) locale, you would create files called format\_en\_GB.properties, exceptions\_en\_GB.properties, and windows\_en\_GB.properties respectively.

Typically, locale resolution is managed by the surrounding environment of the application. In this example, the locale against which (British) messages will be resolved is specified manually.

*# in exceptions\_en\_GB.properties*

argument.required=Ebagum lad, the {0} argument is required, I say, required.

**public** **static** **void** main(**final** String[] args) {

MessageSource resources = **new** ClassPathXmlApplicationContext("beans.xml");

String message = resources.getMessage("argument.required",

**new** Object [] {"userDao"}, "Required", Locale.UK);

System.out.println(message);

}

The resulting output from the running of the above program will be...

Ebagum lad, the 'userDao' argument is required, I say, required.

You can also use the MessageSourceAware interface to acquire a reference to any MessageSource that has been defined. Any bean that is defined in an ApplicationContext that implements the MessageSourceAware interface is injected with the application context's MessageSource when the bean is created and configured.

|  |
| --- |
| [Note] |
| As an alternative to *ResourceBundleMessageSource*, Spring provides a *ReloadableResourceBundleMessageSource* class. This variant supports the same bundle file format but is more flexible than the standard JDK based *ResourceBundleMessageSource* implementation. In particular, it allows for reading files from any Spring resource location (not just from the classpath) and supports hot reloading of bundle property files (while efficiently caching them in between). Check out the ReloadableResourceBundleMessageSource javadoc for details. |

### 5.14.2 Standard and Custom Events

Event handling in the ApplicationContext is provided through the ApplicationEvent class and ApplicationListener interface. If a bean that implements the ApplicationListener interface is deployed into the context, every time an ApplicationEvent gets published to the ApplicationContext, that bean is notified. Essentially, this is the standard Observer design pattern. Spring provides the following standard events:

**Table 5.7. Built-in Events**

| **Event** | **Explanation** |
| --- | --- |
| ContextRefreshedEvent | Published when the ApplicationContext is initialized or refreshed, for example, using the refresh() method on the ConfigurableApplicationContext interface. "Initialized" here means that all beans are loaded, post-processor beans are detected and activated, singletons are pre-instantiated, and the ApplicationContext object is ready for use. As long as the context has not been closed, a refresh can be triggered multiple times, provided that the chosen ApplicationContext actually supports such "hot" refreshes. For example, XmlWebApplicationContext supports hot refreshes, but GenericApplicationContext does not. |
| ContextStartedEvent | Published when the ApplicationContext is started, using the start() method on the ConfigurableApplicationContext interface. "Started" here means that all Lifecycle beans receive an explicit start signal. Typically this signal is used to restart beans after an explicit stop, but it may also be used to start components that have not been configured for autostart , for example, components that have not already started on initialization. |
| ContextStoppedEvent | Published when the ApplicationContext is stopped, using the stop() method on the ConfigurableApplicationContext interface. "Stopped" here means that all Lifecycle beans receive an explicit stop signal. A stopped context may be restarted through a start() call. |
| ContextClosedEvent | Published when the ApplicationContext is closed, using the close() method on the ConfigurableApplicationContext interface. "Closed" here means that all singleton beans are destroyed. A closed context reaches its end of life; it cannot be refreshed or restarted. |
| RequestHandledEvent | A web-specific event telling all beans that an HTTP request has been serviced. This event is published after the request is complete. This event is only applicable to web applications using Spring's DispatcherServlet. |

You can also create and publish your own custom events. This example demonstrates a simple class that extends Spring's ApplicationEvent base class:

**public** **class** BlackListEvent **extends** ApplicationEvent {

**private** **final** String address;

**private** **final** String test;

**public** BlackListEvent(Object source, String address, String test) {

**super**(source);

**this**.address = address;

**this**.test = test;

}

*// accessor and other methods...*

}

To publish a custom ApplicationEvent, call the publishEvent() method on an ApplicationEventPublisher. Typically this is done by creating a class that implements ApplicationEventPublisherAware and registering it as a Spring bean. The following example demonstrates such a class:

**public** **class** EmailService **implements** ApplicationEventPublisherAware {

**private** List<String> blackList;

**private** ApplicationEventPublisher publisher;

**public** **void** setBlackList(List<String> blackList) {

**this**.blackList = blackList;

}

**public** **void** setApplicationEventPublisher(ApplicationEventPublisher publisher) {

**this**.publisher = publisher;

}

**public** **void** sendEmail(String address, String text) {

**if** (blackList.contains(address)) {

BlackListEvent event = **new** BlackListEvent(**this**, address, text);

publisher.publishEvent(event);

**return**;

}

*// send email...*

}

}

At configuration time, the Spring container will detect that EmailService implements ApplicationEventPublisherAware and will automatically call setApplicationEventPublisher(). In reality, the parameter passed in will be the Spring container itself; you're simply interacting with the application context via its ApplicationEventPublisher interface.

To receive the custom ApplicationEvent, create a class that implements ApplicationListener and register it as a Spring bean. The following example demonstrates such a class:

**public** **class** BlackListNotifier **implements** ApplicationListener<BlackListEvent> {

**private** String notificationAddress;

**public** **void** setNotificationAddress(String notificationAddress) {

**this**.notificationAddress = notificationAddress;

}

**public** **void** onApplicationEvent(BlackListEvent event) {

*// notify appropriate parties via notificationAddress...*

}

}

Notice that ApplicationListener is generically parameterized with the type of your custom event, BlackListEvent. This means that the onApplicationEvent() method can remain type-safe, avoiding any need for downcasting. You may register as many event listeners as you wish, but note that by default event listeners receive events synchronously. This means the publishEvent() method blocks until all listeners have finished processing the event. One advantage of this synchronous and single-threaded approach is that when a listener receives an event, it operates inside the transaction context of the publisher if a transaction context is available. If another strategy for event publication becomes necessary, refer to the JavaDoc for Spring's ApplicationEventMulticaster interface.

The following example shows the bean definitions used to register and configure each of the classes above:

<bean id="emailService" class="example.EmailService">

<property name="blackList">

<list>

<value>known.spammer@example.org</value>

<value>known.hacker@example.org</value>

<value>john.doe@example.org</value>

</list>

</property>

</bean>

<bean id="blackListNotifier" class="example.BlackListNotifier">

<property name="notificationAddress" value="blacklist@example.org"/>

</bean>

Putting it all together, when the sendEmail() method of the emailService bean is called, if there are any emails that should be blacklisted, a custom event of type BlackListEvent is published. The blackListNotifier bean is registered as an ApplicationListener and thus receives the BlackListEvent, at which point it can notify appropriate parties.

|  |
| --- |
| [Note] |
| Spring's eventing mechanism is designed for simple communication between Spring beans within the same application context. However, for more sophisticated enterprise integration needs, the separately-maintained [Spring Integration](http://springsource.org/spring-integration) project provides complete support for building lightweight, [pattern-oriented](http://www.enterpriseintegrationpatterns.com/), event-driven architectures that build upon the well-known Spring programming model. |

### 5.14.3 Convenient access to low-level resources

For optimal usage and understanding of application contexts, users should generally familiarize themselves with Spring's Resource abstraction, as described in the chapter [Chapter 6, *Resources*](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/resources.html).

An application context is a ResourceLoader, which can be used to load Resources. A Resource is essentially a more feature rich version of the JDK class java.net.URL, in fact, the implementations of the Resource wrap an instance of java.net.URL where appropriate. A Resource can obtain low-level resources from almost any location in a transparent fashion, including from the classpath, a filesystem location, anywhere describable with a standard URL, and some other variations. If the resource location string is a simple path without any special prefixes, where those resources come from is specific and appropriate to the actual application context type.

You can configure a bean deployed into the application context to implement the special callback interface, ResourceLoaderAware, to be automatically called back at initialization time with the application context itself passed in as the ResourceLoader. You can also expose properties of type Resource, to be used to access static resources; they will be injected into it like any other properties. You can specify those Resource properties as simple String paths, and rely on a special JavaBean PropertyEditor that is automatically registered by the context, to convert those text strings to actual Resource objects when the bean is deployed.

The location path or paths supplied to an ApplicationContext constructor are actually resource strings, and in simple form are treated appropriately to the specific context implementation. ClassPathXmlApplicationContext treats a simple location path as a classpath location. You can also use location paths (resource strings) with special prefixes to force loading of definitions from the classpath or a URL, regardless of the actual context type.

### 5.14.4 Convenient ApplicationContext instantiation for web applications

You can create ApplicationContext instances declaratively by using, for example, a ContextLoader. Of course you can also create ApplicationContext instances programmatically by using one of the ApplicationContext implementations.

You can register an ApplicationContext using the ContextLoaderListener as follows:

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

The listener inspects the contextConfigLocation parameter. If the parameter does not exist, the listener uses /WEB-INF/applicationContext.xml as a default. When the parameter does exist, the listener separates the String by using predefined delimiters (comma, semicolon and whitespace) and uses the values as locations where application contexts will be searched. Ant-style path patterns are supported as well. Examples are /WEB-INF/\*Context.xml for all files with names ending with "Context.xml", residing in the "WEB-INF" directory, and /WEB-INF/\*\*/\*Context.xml, for all such files in any subdirectory of "WEB-INF".

### 5.14.5 Deploying a Spring ApplicationContext as a J2EE RAR file

In Spring 2.5 and later, it is possible to deploy a Spring ApplicationContext as a RAR file, encapsulating the context and all of its required bean classes and library JARs in a J2EE RAR deployment unit. This is the equivalent of bootstrapping a standalone ApplicationContext, just hosted in J2EE environment, being able to access the J2EE servers facilities. RAR deployment is a more natural alternative to scenario of deploying a headless WAR file, in effect, a WAR file without any HTTP entry points that is used only for bootstrapping a Spring ApplicationContext in a J2EE environment.

RAR deployment is ideal for application contexts that do not need HTTP entry points but rather consist only of message endpoints and scheduled jobs. Beans in such a context can use application server resources such as the JTA transaction manager and JNDI-bound JDBC DataSources and JMS ConnectionFactory instances, and may also register with the platform's JMX server - all through Spring's standard transaction management and JNDI and JMX support facilities. Application components can also interact with the application server's JCA WorkManager through Spring's TaskExecutor abstraction.

Check out the JavaDoc of the [SpringContextResourceAdapter](http://static.springsource.org/spring-framework/docs/current/javadoc-api/org/springframework/jca/context/SpringContextResourceAdapter.html) class for the configuration details involved in RAR deployment.

For a simple deployment of a Spring ApplicationContext as a J2EE RAR file: package all application classes into a RAR file, which is a standard JAR file with a different file extension. Add all required library JARs into the root of the RAR archive. Add a "META-INF/ra.xml" deployment descriptor (as shown in SpringContextResourceAdapters JavaDoc) and the corresponding Spring XML bean definition file(s) (typically "META-INF/applicationContext.xml"), and drop the resulting RAR file into your application server's deployment directory.

|  |
| --- |
| [Note] |
| Such RAR deployment units are usually self-contained; they do not expose components to the outside world, not even to other modules of the same application. Interaction with a RAR-based ApplicationContext usually occurs through JMS destinations that it shares with other modules. A RAR-based ApplicationContext may also, for example, schedule some jobs, reacting to new files in the file system (or the like). If it needs to allow synchronous access from the outside, it could for example export RMI endpoints, which of course may be used by other application modules on the same machine. |

## 5.15 The BeanFactory

The BeanFactory provides the underlying basis for Spring's IoC functionality but it is only used directly in integration with other third-party frameworks and is now largely historical in nature for most users of Spring. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purposes of backward compatibility with the large number of third-party frameworks that integrate with Spring. Often third-party components that can not use more modern equivalents such as @PostConstruct or @PreDestroy in order to remain compatible with JDK 1.4 or to avoid a dependency on JSR-250.

This section provides additional background into the differences between the BeanFactory and ApplicationContext and how one might access the IoC container directly through a classic singleton lookup.

### 5.15.1 BeanFactory or ApplicationContext?

Use an ApplicationContext unless you have a good reason for not doing so.

Because the ApplicationContext includes all functionality of the BeanFactory, it is generally recommended over the BeanFactory, except for a few situations such as in an Applet where memory consumption might be critical and a few extra kilobytes might make a difference. However, for most typical enterprise applications and systems, the ApplicationContext is what you will want to use. Spring 2.0 and later makes heavy use of the [BeanPostProcessor extension point](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-extension-bpp) (to effect proxying and so on). If you use only a plain BeanFactory, a fair amount of support such as transactions and AOP will not take effect, at least not without some extra steps on your part. This situation could be confusing because nothing is actually wrong with the configuration.

The following table lists features provided by the BeanFactory and ApplicationContext interfaces and implementations.

**Table 5.8. Feature Matrix**

| **Feature** | **BeanFactory** | **ApplicationContext** |
| --- | --- | --- |
| Bean instantiation/wiring | Yes | Yes |
| Automatic BeanPostProcessor registration | No | Yes |
| Automatic BeanFactoryPostProcessor registration | No | Yes |
| Convenient MessageSource access (for i18n) | No | Yes |
| ApplicationEvent publication | No | Yes |

To explicitly register a bean post-processor with a BeanFactory implementation, you must write code like this:

ConfigurableBeanFactory factory = **new** XmlBeanFactory(...);

*// now register any needed BeanPostProcessor instances*

MyBeanPostProcessor postProcessor = **new** MyBeanPostProcessor();

factory.addBeanPostProcessor(postProcessor);

*// now start using the factory*

To explicitly register a BeanFactoryPostProcessor when using a BeanFactory implementation, you must write code like this:

XmlBeanFactory factory = **new** XmlBeanFactory(**new** FileSystemResource("beans.xml"));

*// bring in some property values from a Properties file*

PropertyPlaceholderConfigurer cfg = **new** PropertyPlaceholderConfigurer();

cfg.setLocation(**new** FileSystemResource("jdbc.properties"));

*// now actually do the replacement*

cfg.postProcessBeanFactory(factory);

In both cases, the explicit registration step is inconvenient, which is one reason why the various ApplicationContext implementations are preferred above plain BeanFactory implementations in the vast majority of Spring-backed applications, especially when using BeanFactoryPostProcessors and BeanPostProcessors. These mechanisms implement important functionality such as property placeholder replacement and AOP.

### 5.15.2 Glue code and the evil singleton

It is best to write most application code in a dependency-injection (DI) style, where that code is served out of a Spring IoC container, has its own dependencies supplied by the container when it is created, and is completely unaware of the container. However, for the small glue layers of code that are sometimes needed to tie other code together, you sometimes need a singleton (or quasi-singleton) style access to a Spring IoC container. For example, third-party code may try to construct new objects directly (Class.forName() style), without the ability to get these objects out of a Spring IoC container. If the object constructed by the third-party code is a small stub or proxy, which then uses a singleton style access to a Spring IoC container to get a real object to delegate to, then inversion of control has still been achieved for the majority of the code (the object coming out of the container). Thus most code is still unaware of the container or how it is accessed, and remains decoupled from other code, with all ensuing benefits. EJBs may also use this stub/proxy approach to delegate to a plain Java implementation object, retrieved from a Spring IoC container. While the Spring IoC container itself ideally does not have to be a singleton, it may be unrealistic in terms of memory usage or initialization times (when using beans in the Spring IoC container such as a Hibernate SessionFactory) for each bean to use its own, non-singleton Spring IoC container.

Looking up the application context in a service locator style is sometimes the only option for accessing shared Spring-managed components, such as in an EJB 2.1 environment, or when you want to share a single ApplicationContext as a parent to WebApplicationContexts across WAR files. In this case you should look into using the utility class [ContextSingletonBeanFactoryLocator](http://static.springsource.org/spring/docs/current/api/org/springframework/context/access/ContextSingletonBeanFactoryLocator.html) locator that is described in this [SpringSource team blog entry](http://blog.springsource.com/2007/06/11/using-a-shared-parent-application-context-in-a-multi-war-spring-application/).

[[1]](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#d5e1144)See [Background](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/overview.html#background-ioc)

[[2]](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#d5e1977)See [Section 5.4.1, “Dependency injection”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-factory-collaborators)

|  |  |  |
| --- | --- | --- |
| [Prev](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/spring-core.html) | [Up](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/spring-core.html) | [Next](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/resources.html) |
| Part III. Core Technologies | [Home](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/index.html) | 6. Resources |

# Spring Boot

# Understanding the Basics of Spring vs. Spring Boot

Over the past few years, due to added functionalities, the Spring framework has become increasingly complex. It requires going through a lengthy procedure in order to start a new Spring project. To avoid starting from scratch and save time, Spring Boot has been introduced. This uses the Spring framework as a foundation.

In this blog, we are going to focus on the roles of Spring and Spring Boot in solving various problems and how they are different from one another. Let’s get started!

To understand the difference between Spring and the Spring Boot framework, let us consider an application that allows requests from the various browser. Below is the flow we want to consider:

**Browser <> RESTful layer <> Middle layer <> Database**

## ****Spring Framework****

Spring is one of the most widely used Java EE Frameworks for building applications. For the [Java platform](https://www.sprintzeal.com/blog/technology/top-7-trending-technology-skills-to-watch-out-in-2018), the Spring framework provides an elaborate programming and configuration model. It aims to simplify the Java EE development and helps developers be more productive at work. It can be used at any kind of deployment platform. It takes into account the rising needs of today’s businesses and strives to fulfill them.

Unlike other frameworks, Spring focuses on several areas of an application and provides a wide range of features.

One of the major features of the Spring framework is the dependency injection. It helps make things simpler by allowing us to develop loosely coupled applications.

## ****Spring Boot****

While the Spring framework focuses on providing flexibility to you, Spring Boot aims to shorten the code length and provide you with the easiest way to develop a web application. With annotation configuration and default codes, Spring Boot shortens the time involved in developing an application. It helps create a stand-alone application with less or almost zero-configuration.

Autoconfiguration is a special feature in Spring Boot. It automatically configures a class based on that requirement. Let us take a quick look at some of the features and benefits of these frameworks:

## ****Benefits of the Spring Framework****

•    The Spring framework can be used for all layers of implementation in the development of an application.

•    Considering its POJO model, it is a very lightweight framework.

•    It allows loose coupling and easy testability.

•    It supports declarative programming.

•    It is capable of eliminating the formation of singleton and factory classes.

•    It supports both XML and annotation configurations.

•    It provides middleware services.

Despite having several benefits in the Spring framework, what led to the emergence of Spring Boot?

Spring Boot helps in the easy usage of the Spring Framework by simplifying it to a great extent. Spring provides a loosely coupled application — this is a great feature. However, when there are several loosely coupled blocks, keeping track of them becomes a tedious and messy task. This is where Spring Block comes into the picture and helps simplify things by offering no configuration feature. It helps you get started with minimal effort and even provides externalized configuration.

## ****Benefits of Spring Boot****

•    [Spring Boot](https://dzone.com/articles/spring-vs-spring-boot) doesn’t require you to deploy WAR files.

•    It creates stand-alone applications.

•    It helps embed Tomcat, Jetty, or Undertow directly.

•    It doesn’t require XML configuration.

•    It aims to reduce the LOC.

•    It offers production ready features.

•    It is easier to launch.

•    Easier customization and management.

Therefore, Spring Boot is a Spring-based production-ready project initializer. With features like auto-configuration, it saves you from writing lengthy code and helps you avoid unnecessary configuration.

While the Spring Framework offers you features like dependency injection or IOC and handles transactions, it also acts as a foundation for other Spring frameworks. The best example for this is Spring boot. Spring Boot uses the Spring Framework as a foundation and improvises on it. It simplifies Spring dependencies and runs applications straight from a command line. It also **doesn’t require an application container**. Spring Boot mostly helps in monitoring several components and configures them externally.

All in all, the Spring framework has made a significant contribution and continues to do so. With the many features mentioned above, the Spring framework is always a great choice for developers. However, it is highly beneficial when used alongside Spring Boot. The added advantages that come with Spring Boot are of great value to the developers as they offer completion of projects with minimal efforts. To all the problems that arise from the Spring framework, Spring Boot is the solution.

# Hibernate

# Web Services

## What is Web Services, Web Services Introduction

[Web Services](https://www.java4s.com/web-services/) » on Jul 6, 2014 [**{ 23 Comments }**](https://www.java4s.com/web-services/what-is-web-services-web-services-introduction/#comments) By Sivateja

What is Web Services ? Over the internet, you might have seen different kinds of definitions for Web services. My definition will almost resembles them 🙂 Web Services, the name it self indicates that its a service which is available over the Web, that’s it. As an example you can consider Java4s.com, When ever you hit the URL in the web browser it will gives you some output in HTML format, you can also consider this as a Web service.  With web services, we can communicate different applications on different platforms, i mean a java application in Windows platform can easily communicate with the application developed using .net/php in Linux operation system.

## Understanding SOAP and REST

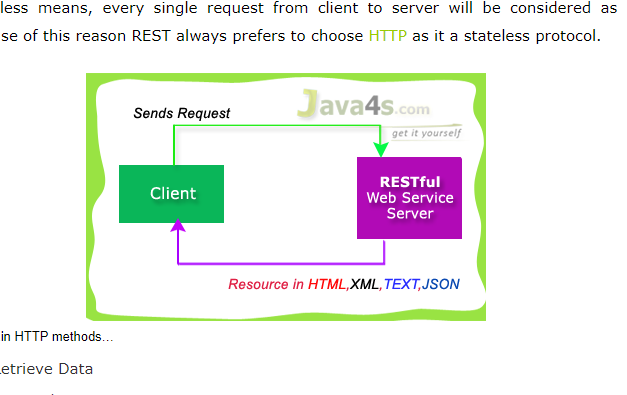
Web Services are mainly of 2 types, **SOAP** [Simple Object Access Protocol] and **REST** [Representational state transfer] based services. We have different type of specifications to implement SOAP and REST services. I believe so far you might be in confusion with these kind keywords like, JAX-RS, JAX-WS, RESTful, SOAP, Apache Axis2, Apache CXF bla bla…  Let me try to bring you out of them.

* JAX-RS provides the implementation of RESTful web services,  JAX-RS is a **specification** for RESTful Web Services with Java and it is given by Sun.  Since it is a specification, other frameworks can be written to implement these specifications, and that includes Jersey from Oracle, Resteasy from Jboss, CXF from Apache bla bla.
* **JAX-WS, Apache Axis2** provides the implementation for SOAP
* **Apache CXF** provides implementation for SOAP and RESTful services both.

## RESTful

What ever **the data/response we will get from the server** is known as ***Resource*** [remember this point], Each resource can be accessed by its URI’s.  We can get the resource from RESTful service in different formats like, **HTML,XML,JSON,TEXT,PDF** and in the Image formats as well, but in real time we mainly we will prefer **JSON**.  REST guidelines always talks about **stateless** communication between client and the Server.  Stateless means, every single request from client to server will be considered as a fresh request. Because of this reason REST always prefers to choose HTTP as it a stateless protocol.

RESTful used 4 main HTTP methods…



* **GET** – Retrieve Data
* **POST**– Create/Insert Data
* **PUT**– Update Data
* **DELETE**– Delete Data

Generally we will prefer RESTful Services in these scenarios…

* If clients require **caching**, means if you have limited bandwidth
* If you want every thing to be **stateless** [ I have already explained about stateless ]

But SOAP gives the output only in XML format.   Hope you are good now 🙂 by the way we are going to use Jersey to implement JAX-RS specifications.

# Dev-Ops

# Data Structure

<https://www.java67.com/search/label/data%20structure%20and%20algorithm?&max-results=3> for datastructure

## External Sorting

External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive). External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub-files are combined into a single larger file.

One example of external sorting is the external merge sort algorithm, which sorts chunks that each fit in RAM, then merges the sorted chunks together. We first divide the file into **runs** such that the size of a run is small enough to fit into main memory. Then sort each run in main memory using merge sort sorting algorithm. Finally merge the resulting runs together into successively bigger runs, until the file is sorted.

Prerequisite for the algorithm/code:  
[MergeSort](http://geeksquiz.com/merge-sort/) : Used for sort individual runs (a run is part of file that is small enough to fit in main memory)  
[Merge K Sorted Arrays](https://www.geeksforgeeks.org/merge-k-sorted-arrays/) : Used to merge sorted runs.

## Merge Sort in Java - Algorithm Example and Tutorial

The merge sort algorithm is a divide and conquers algorithm. In the divide and conquer paradigm, a problem is broken into smaller problems where each small problem still retains all the properties of the larger problem -- except its size. To solve the original problem, each piece is solved individually; then the pieces are merged back together. For example, imagine you have to sort an array of 200 elements using the [bubble sort](http://www.java67.com/2012/12/bubble-sort-in-java-program-to-sort-integer-array-example.html) algorithm. Since selection sort takes O(n^2) time, it would take about 40,000-time units to sort the array. Now imagine splitting the array into ten equal pieces and sorting each piece individually still using selection sort. Now it would take 400-time units to sort each piece; for a grand total of 10\*400 = 4000.  
  
Once each piece is sorted, merging them back together would take about 200-time units; for a grand total of 200+4000 = 4,200. Clearly, 4,200 is an impressive improvement over 40,000.  
  
Now think bigger. Imagine splitting the original array into groups of two and then sorting them. In the end, it would take about 1,000-time units to sort the array.  
  
That's how merge sort works. It makes sorting a big array easy and hence its suitable for large integer and string arrays. Time Complexity of the mergesort algorithm is the same in the best, average and worst case and it's equal to O(n\*log(n))  
  
Btw, if you are new to Algorithms and Data Structure and not familiar with an essential sorting and searching algorithms like quicksort, binary search, level order search etc  then I suggest you go through a good, comprehensive online course like [**Data Structures and Algorithms: Deep Dive Using Java**](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https%3A%2F%2Fwww.udemy.com%2Fdata-structures-and-algorithms-deep-dive-using-java%2F) to learn the basics first.

## Sorting Array using Merge Sort Algorithm:

You have given an unordered list of integers (or any other objects e.g. String), You have to rearrange the integers or objects in their natural order.  
  
Sample Input: {80, 50, 30, 10, 90, 60, 0, 70, 40, 20, 5}  
Sample Output: {0, 10, 20, 30, 40, 50, 50, 60, 70, 80, 90}

**import** java.util.Arrays;

/\*

\* Java Program to sort an integer array using merge sort algorithm.

\*/

**public** class Main {

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

System.out.println("mergesort");

**int**[] **input** **=** { 87, 57, 370, 110, 90, 610, 02, 710, 140, 203, 150 };

System.out.println("array before sorting");

System.out.println(Arrays.**toString**(**input**));

// sorting array using MergeSort algorithm

mergesort(**input**);

System.out.println("array after sorting using mergesort algorithm");

System.out.println(Arrays.**toString**(**input**));

}

/\*\*

\* Java function to sort given array using merge sort algorithm

\*

\* @param input

\*/

**public** **static** **void** mergesort(**int**[] **input**) {

mergesort(**input**, 0, **input**.**length** **-** 1);

}

/\*\*

\* A Java method to implement MergeSort algorithm using recursion

\*

\* @param input

\* , integer array to be sorted

\* @param start

\* index of first element in array

\* @param end

\* index of last element in array

\*/

**private** **static** **void** mergesort(**int**[] **input**, **int** **start**, **int** end) {

// break problem into smaller structurally identical problems

**int** mid **=** (**start** **+** end) **/** 2;

**if** (**start** < end) {

mergesort(**input**, **start**, mid);

mergesort(**input**, mid **+** 1, end);

}

// merge solved pieces to get solution to original problem

**int** i **=** 0, **first** **=** **start**, **last** **=** mid **+** 1;

**int**[] tmp **=** **new** **int**[end **-** **start** **+** 1];

**while** (**first** <**=** mid **&&** **last** <**=** end) {

tmp[i**++**] **=** **input**[**first**] < **input**[**last**] **?** **input**[**first++**] **:** **input**[**last++**];

}

**while** (**first** <**=** mid) {

tmp[i**++**] **=** **input**[**first++**];

}

**while** (**last** <**=** end) {

tmp[i**++**] **=** **input**[**last++**];

}

i **=** 0;

**while** (**start** <**=** end) {

**input**[**start++**] **=** tmp[i**++**];

}

}

}

Output

mergesort

array before sorting

[87, 57, 370, 110, 90, 610, 2, 710, 140, 203, 150]

array after sorting using mergesort algorithm

[2, 57, 87, 90, 110, 140, 150, 203, 370, 610, 710]

You can see that the array is now sorted. The algorithm we have used is a recursive implementation of merge sort and it's also a [stable sorting algorithm](https://javarevisited.blogspot.com/2017/06/difference-between-stable-and-unstable-algorithm.html) I mean it maintains the original order of elements in case of a tie.  
  
Anyway, if you haven't got it yet that how merge sort algorithm works, you can also check out the [**Algorithms and Data Structures - Part 1 and 2**](https://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fads-part1) course on Pluralsight which explains key sorting and searching algorithms in a very nice way. It also covers essential data structures like linked list, array, hash table, binary tree etc.

## Introduction to Data Structures

<https://www.codesdope.com/course/data-structures-introduction/>

Data Structures and Algorithms (or DSA) is the most important course of any Computer Science program. In this course, we will deal with different data structures, their applications, running times, etc. If you don’t know how to analyze the running time of code, you can read the first 7 chapters of the [Algorithm Course](https://www.codesdope.com/course/algorithms-introduction/). It is also recommended that you first read those chapters before proceeding with this course.

In this very first chapter of data structures, we will focus on learning what basically is a data structure and why do we need it. So, let’s start.

## What is a data structure?

A data structure is a way we store and organize our data. For example, think about organizing books in a room, we can keep those books on a shelf, or make a stack of them on a table or even just put them randomly anywhere in the room.

Thus, we have different options to organize books in a room or in different words, we have different structures to keep books. In computers also, we have a similar scenario i.e., we can organize our data in the way we want and these different ways of organizing data are different data structures.

For example, an array is a type of data structure which we learn while learning basic programming languages. It is the most basic data structure and stores different data at different indices.

There are many different data structures which are generally used. Many programming languages also provide pre-built libraries for many data structure. But in this course, every discussed data structure is made from scratch.

## Do we really need to worry about how our data is stored?

One can keep a frequently used book at the bottom of the pile of books and can access it with a little difficulty but it would make a lot more sense to keep frequently used books on the shelf to access them with ease.

In computers also, the choice of the data structure depends upon the task we are going to perform. For example, if we have a constant number of data and accessing the data in the least time is our priority, then an array is a suitable data structure because it can return the data at an index in constant time (O(1)O(1)).

But imagine a task in which we need to frequently insert some new data between two data. In that case, using an array will lead to shifting the elements of the array or even making a new array of different size if the array is not large enough.

So, a data structure in which the task of inserting some new data between two data is done in the least time would be suitable for this purpose.

The point is that we can complete a task using any data structure but a suitable data structure for a task not only reduces the programmer's effort but also saves a lot of computational time and space.

For example, imagine searching for a city in the list of all the cities of a country. If the desired city is at the last of the list, we will end up iterating over the entire list.

But if we organize all the cities under the state in which they lie and we know the state, it would be a much quicker process to search the city

## Can’t I just use libraries instead of making a data structure from scratch?

The first point is that you need to at least understand the working of the data structure even to use a library. So, assuming that a person has an understanding of the data structure being used and the library provides exactly what the person needs, of course, a library can be used.

Even though we can use a library for simpler data structures but we often need a more complex data structure which is made using simpler data structures and existing libraries of them doesn’t always provide exactly what we need and we end up writing our own data structure from scratch. Sometimes this also happens with simpler data structures and we also make them from scratch to suit our need.

## Should I also be concerned with the choice of language for the implementation of data structure?

The implementation in a language like C is done with the help of structure, pointer, etc. Whereas in an objected oriented language like Java, it is done with classes and objects and the idea remains the same as long as the language is an object-oriented one. So, the implementation will change with the "**type**" of the language we are using.

In this course, we are going to implement every data structure in three different languages - C/C++, Java and Python, you can proceed with the language you know.

## What this course teaches me?

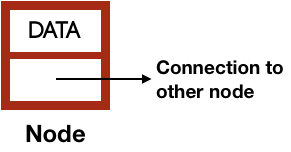
In this course, you will learn the basic concepts of different data structures, their applications and their implementation in different languages. We will also focus on the running time of different processes like inserting data, searching data, etc. in a data structure. At the end of this course, you will have knowledge of different data structures and you can use this knowledge to create a new data structure or modify an existing one according to your need.

## Linked Lists

So far, we know that a data structure is a way to keep our data, and a linked list is the first data structure we are going to study in this course. Breaking the name "**Linked List**", it is a list of elements (containing data) in which the elements are linked together.

linked list

As you can see in the above picture, each data is connected (or linked) to a different data and thus, forming a linear list of data, and this is a linked list. Since data in a linked list are stored in a linear fashion, a linked list is a linear data structure. There are also non-linear data structures like trees, graphs, etc. which we are going to study further in this course.

Till now, you know that we have to connect data of a linked list and to do so, we put our data in nodes and these nodes are connected in the desired fashion.

So before going further, let's discuss about nodes.

## Node

A node can be viewed as a container or a box which contains data and other information in it. Nodes are connected (or organized) in a specific way to make data structures like linked lists, trees, etc.

In the above picture, each node has two parts, one stores the data and another is connected to a different node. The last node is not connected to any other node and thus, its connection to the next node is null.

In a linked list, a node is connected to a different node forming a chain of nodes.

Thus to make a linked list, we first need to make a node which should store some data into it and also a link to another node.

There can be different ways to make this node in different languages, we are going to discuss the making of the node in C, Java and Python

**class** Node {

**public** **int** data;

**public** Node next;

}

You can see that we have created a class named Node to represent a node and the class has two members - the first is the data (an integer) which our node is going to store and the other is another node which is the next node.

As we have created our class which meets our requirement of a node, let’s create two objects of this Node class and connect the next of the first node with the second one.

**class** Node {

**public** **int** data;

**public** Node next;

**public** Node(**int** d) {

data = d;

}

}

**class** List {

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

Node a, b;

a = **new** Node(10);

b = **new** Node(20);

a.next = b;

b.next = **null**;

System.out.print(a.data+"\n"+a.next.data+"\n");

}

}

Node a, b; → Here, we are creating two nodes (or two objects of the Node class).

a.next = b; → We are storing *b* in next of *a*. So, *next* of *a* is now linked to the node *b*.

Since *b* is the last node (there is no node to point next of *b* to), we are making *next* of *b* null - b.next = null;.

## Making and Traversing a Linked List

In the above examples, we have made two nodes *a* and *b* and we were able to access the node *b* with a.next. Suppose there are few more nodes, we can easily access them if we have access to the first node. This is explained in the picture given below.



Generally, we call the first node of a linked list the "head" of the linked list, and we always keep the access of the head of the linked list so that we have access to all the nodes of a linked list.

### **Traversing a Linked List**

Traversing is visiting all the nodes of a linked list. As stated above, we always keep a record of the head of a linked list. We also know that the last node of a linked list is null.



So, we start a loop from the head of the linked list and end it when the node is null.



Let's make a function to traverse over all the nodes of a linked list. We will pass the linked list (L) to it - TRAVERSE(L).

Now, we make a temporary pointer and point it to the head of the linked list (L) - temp = L.head.

We have a pointer pointing to the head of the linked list, thus we can start a loop and end it when this pointer will point to null (last element) i.e., while(temp != null).

At the end of each iteration, we have to just point the *temp* pointer to the next of the node i.e., temp = temp.next.

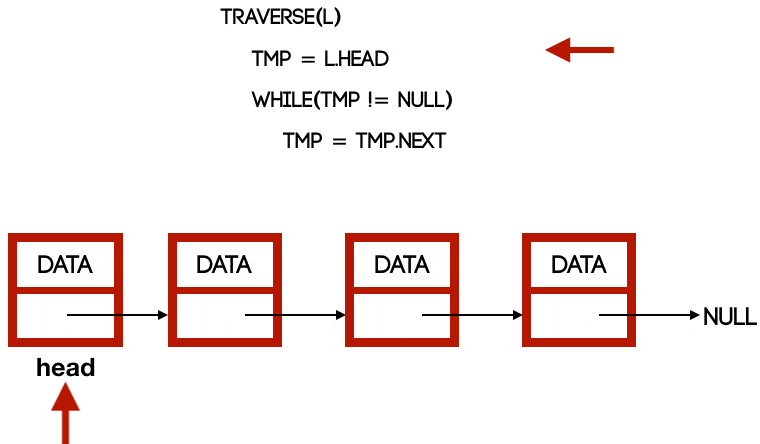
**TRAVERSE(L)**

**temp = L.head**

**while(temp != null)**

**temp = temp.next**

So, temp = temp.next will make the *temp* point the next node in each iteration and the loop will run until the last node (until *temp* is not null).



Till now, we know what a linked list is, what a node is and how to traverse over each node of a linked list. Let's learn about adding nodes to a linked list so that we can make a linked list.

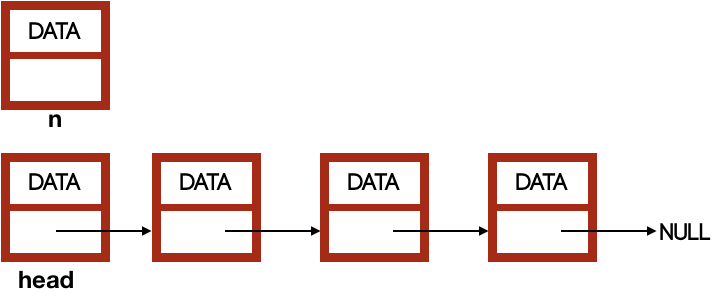
## Inserting Nodes in a Linked List

There can be three different positions where we can insert a new node in a linked list:

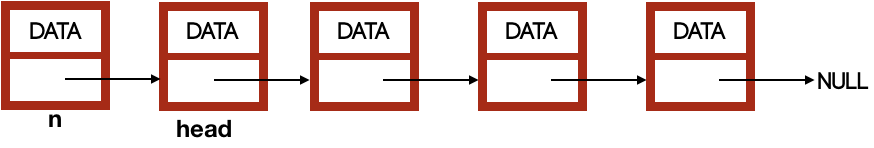
* At the beginning of the list.
* At the end of the list.
* Anywhere except the above-mentioned positions.

### **Inserting a New Node at the Beginning of a Linked List**

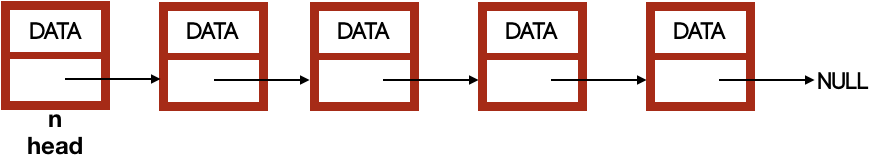
We start by passing the node (n) and the linked list (L) to a function i.e., INSERT\_AT\_BEGINNING(L, n).



After this, we point the next of the node to the head of the linked list - n.next = L.head.



Since the head should always point to the first element of the linked list, so we change the head to point to the new node n i.e, L.head = n.



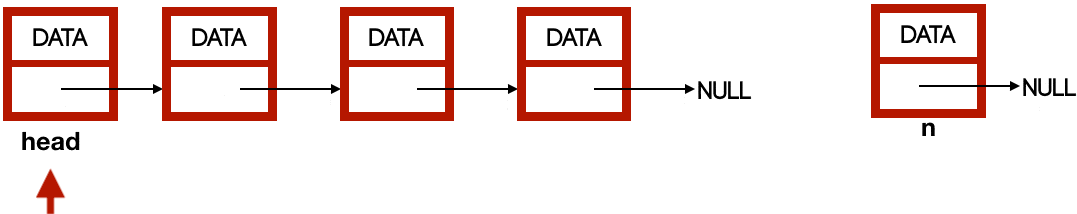
**INSERT\_AT\_BEGINNING(L, n)**

**n.next = L.head**

**L.head = n**

### **Inserting a New Node at the End of a Linked List.**

To insert a node at the end of a linked list, we just iterate to the last of the linked list and add a new node there. This is described in the picture given below.



We start by passing the linked list and the node to the function - INSERT\_AT\_LAST(L, n).

Our next task is to iterate to the last of the linked list.

temp = L.head  
while(temp.next != null)  
    temp = temp.next

After this, we just need to add the node there.

**INSERT\_AT\_LAST(L, n)**

**temp = L.head**

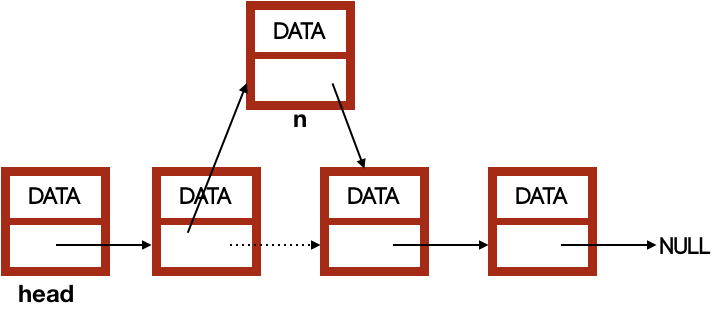
**while(temp.next != null)**

**temp = temp.next**

**temp.next = n**

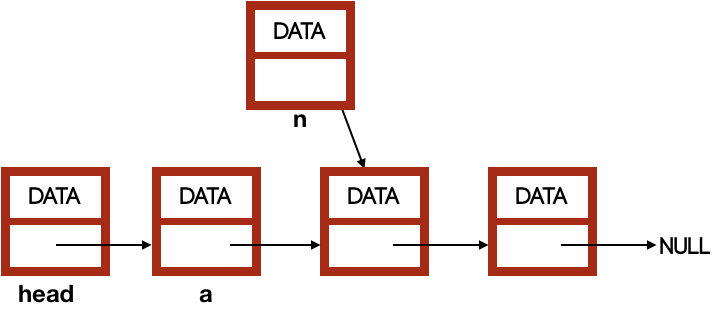
### **Inserting a New Node in the Middle of a Linked List**

To insert a new node in the middle of a linked list, we need to break the existing links and create new links. This will be clear from the picture given below.



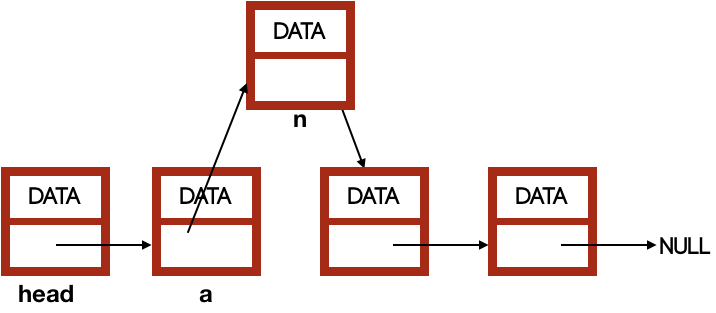
So, we start by passing the node to be inserted (n) and the node after which we are going to insert this node (a) i.e., INSERT\_NODE\_AFTER(n, a).

We point *next* of the new node (n) to *next* of the node *a*.

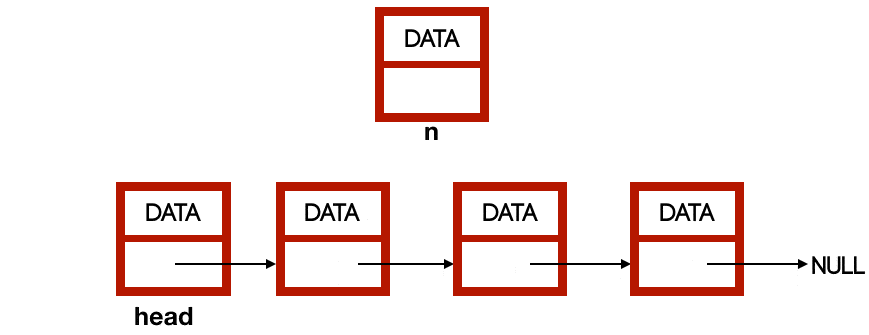


n.next = a.next

After this, we point *next* of the *a* to the new node.



a.next = n



**INSERT\_NODE\_AFTER(n, a)**

**n.next = a.next**

**a.next = n**

Now, we are able to create a linked list and also add nodes to it. Let’s complete this chapter by learning to delete a node from a linked list.

## Deleting a Node from a Linked List

We delete any node of a linked list by connecting the predecessor node of the node to be deleted by the successor node of the node. For example, if we have a linked list a → b → c, then to delete the node 'b', we will connect 'a' to 'c' i.e., a → c. But this will make the node 'b' inaccessible and this type of inaccessible nodes are called **garbage**.

We might need to clean this garbage ourself in some languages like C by using the [free function](https://www.codesdope.com/c-dynamic-memory/) while some languages like Java does it automatically. In the pseudocode, we will assume that a language does it automatically. However, you can see the full code in C, Java and Python at the end of this chapter.

**DEL(L, n)**

**tmp = L.head**

**if tmp == n // node to be deleted is head**

**L.head = n.next**

**else // node to be deleted is not head**

**while(tmp != null)**

**if tmp.next == n**

**tmp.next = n.next //linking**

**break**

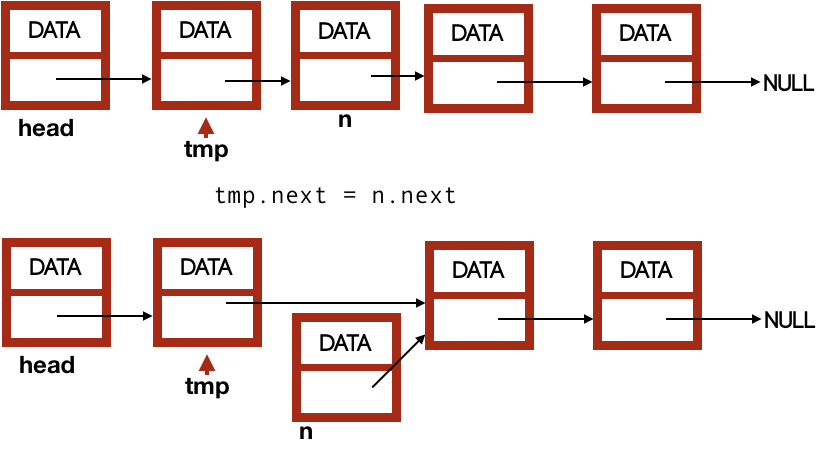
**tmp = tmp.next**

We will first iterate to the node previous to the node *n*. So, we will first start from the head of the linked list - tmp = L.head.

If the node we are going to delete is the head of the linked list (if tmp == n), then we will just update the head pointer - L.head = n.next.



Otherwise, we will iterate to the node previous to the node *n* - if tmp.next == n and then link the node previous of the node to be deleted to the next of it (tmp.next = n.next).



* **C**

* **Python**

* **Java**

***#include*** *<stdio.h>*

***#include*** *<stdlib.h>*

***typedef******struct*** *node {*

***int*** *data;*

***struct*** *node \*next;*

*}node;*

***typedef******struct*** *linked\_list {*

***struct*** *node \*head;*

*}linked\_list;*

*//to make new node*

*node\* new\_node(****int*** *data) {*

*node \*z;*

*z = malloc(****sizeof****(****struct*** *node));*

*z->data = data;*

*z->next = NULL;*

***return*** *z;*

*}*

*//to make a new linked list*

*linked\_list\* new\_linked\_list(****int*** *data) {*

*node \*a; //new node for head of linked list*

*a = new\_node(data);*

*linked\_list \*l = malloc(****sizeof****(linked\_list)); //linked list*

*l->head = a;*

***return*** *l;*

*}*

***void*** *traversal(linked\_list \*l) {*

*node \*temp = l->head; //temporary pointer to point to head*

***while****(temp != NULL) { //iterating over linked list*

*printf("%d\t", temp->data);*

*temp = temp->next;*

*}*

*printf("\n");*

*}*

*//new node before head*

***void*** *insert\_at\_beginning(linked\_list \*l, node \*n) {*

*n->next = l->head;*

*l->head = n;*

*}*

*//insert new node at last*

***void*** *insert\_at\_last(linked\_list \*l, node \*n) {*

*node \*temp = l->head;*

***while****(temp->next != NULL) {*

*temp = temp->next;*

*}*

*temp->next = n;*

*}*

*//function to insert a node after a node*

***void*** *insert\_node\_after(node \*n, node \*a) {*

*n->next = a->next;*

*a->next = n;*

*}*

*//function to delete*

***void*** *del(linked\_list \*l, node \*n) {*

*node \*temp = l->head;*

***if****(temp == n) { //node to be deleted is head*

*l->head = n->next;*

*free(n);*

*}*

***else*** *{ //node to be deleted is not head*

***while****(temp != NULL) {*

***if****(temp->next == n) { //node previous to node to be deleted*

*temp->next = n->next;*

*free(n);*

***break****; //breaking the loop after deleting the node*

*}*

*temp = temp->next;*

*}*

*}*

*}*

***int*** *main() {*

*linked\_list \*l = new\_linked\_list(10);*

*node \*a, \*b, \*c; //new nodes to insert in linekd list*

*a = new\_node(20);*

*b = new\_node(50);*

*c = new\_node(60);*

*//connecting to linked list*

*/\**

*---- ---- ---- ----*

*|head|-->| a |-->| b |-->| c |-->NULL*

*|\_\_\_\_| |\_\_\_\_| |\_\_\_\_| |\_\_\_\_|*

*\*/*

*l->head->next = a;*

*a->next = b;*

*b->next = c;*

*traversal(l);*

*node \*z;*

*z = new\_node(0);*

*insert\_at\_beginning(l, z);*

*z = new\_node(-10);*

*insert\_at\_beginning(l, z);*

*z = new\_node(100);*

*insert\_at\_last(l, z);*

*z = new\_node(30);*

*insert\_node\_after(z, a);*

*z = new\_node(40);*

*insert\_node\_after(z, a->next);*

*z = new\_node(500);*

*insert\_node\_after(z, a->next->next);*

*traversal(l);*

*del(l, l->head);*

*del(l, z);*

*traversal(l);*

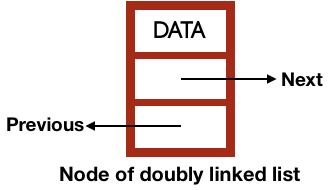
***return*** *0;*

*}*

## Analysis of Linked List

A linked list is a linear data structure like an array but the size of the linked can be changed, unlike an array. However, to access an element from a linked list, we might traverse to the entire list, so accessing an element takes O(n)O(n) time. Inserting is also O(n)O(n) operation because we need to traverse the entire list when the node to be operated is at the last of the linked list.

We learned about a singly linked list i.e., we had a single link between two nodes. We can also have one more link between nodes to point to the previous node of any node which is called a doubly linked list. We are going to study about it in the [next chapter](https://www.codesdope.com/course/data-structures-doubly-linked-lists/).



You can always check the articles in further reading to learn more about any topic. Also, make sure to download the [BlogsDope app](https://play.google.com/store/apps/details?id=com.blogsdope) to stay tuned with us.

# Sorting Algorithm

### Swapping two variable(a,b):

#### Using temp var

Temp=a;

A=b;

B=temp;

#### Without using temp variable:

A=a-b;

B=a+b;

A=b-a;

## Selection Sort

he selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.  
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Following example explains the above steps:

arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]

// and place it at beginning

**11** 25 12 22 64

// Find the minimum element in arr[1...4]

// and place it at beginning of arr[1...4]

11 **12** 25 22 64

// Find the minimum element in arr[2...4]

// and place it at beginning of arr[2...4]

11 12 **22** 25 64

// Find the minimum element in arr[3...4]

// and place it at beginning of arr[3...4]

11 12 22 **25** 64

**import** java.util.Scanner;

**public** **class** SelectionSort {

**public** **int**[] selectionSort(**int**[] a,**int** n)

{

**int** c;

**for**(**int** i=0;i<n-1;i++)

{

**for**(**int** j=i+1;j<n;j++)

{

**if**(a[i]>a[j])

{

c=a[j];

a[j]=a[i];

a[i]=c;

}

}

}

**return** a;

}

**public** **void** printNo(**int**[] a, **int** n)

{

**for**(**int** i=0;i<n;i++)

System.***out***.println(a[i]);

}

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

SelectionSort ss=**new** SelectionSort();

// **TODO** Auto-generated method stub

Scanner sc=**new** Scanner(System.***in***);

System.***out***.println("How many no you want to Sort");

**int** n=sc.nextInt();

**int** a[]=**new** **int**[n];

**for**(**int** i=0;i<n;i++)

{

**int** x=sc.nextInt();

a[i]=x;

}

**int** b[]=ss.selectionSort(a,n);

ss.printNo(b,n);

}

}

## Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

**Example:**  
**First Pass:**  
( **5** **1** 4 2 8 ) –> ( **1** **5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.  
( 1 **5** **4** 2 8 ) –>  ( 1 **4** **5** 2 8 ), Swap since 5 > 4  
( 1 4 **5** **2** 8 ) –>  ( 1 4 **2** **5** 8 ), Swap since 5 > 2  
( 1 4 2 **5** **8** ) –> ( 1 4 2 **5** **8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

**Second Pass:**  
( **1** **4** 2 5 8 ) –> ( **1** **4** 2 5 8 )  
( 1 **4** **2** 5 8 ) –> ( 1 **2** **4** 5 8 ), Swap since 4 > 2  
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) –>  ( 1 2 4 **5** **8** )  
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

**Third Pass:**  
( **1** **2** 4 5 8 ) –> ( **1** **2** 4 5 8 )  
( 1 **2** **4** 5 8 ) –> ( 1 **2** **4** 5 8 )  
( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) –> ( 1 2 4 **5** **8** )

**import** java.util.Scanner;

**public** **class** BubbleSort {

**public** **int**[] bubbleSort(**int**[] a,**int** n)

{

**int** c;

**for**(**int** i=n-1;i>0;i--)

{

**for**(**int** j=1;j<i;j++)

{

**if**(a[j]>a[j+1])

{

c=a[j+1];

a[j+1]=a[j];

a[j]=c;

}

}

}

**return** a;

}

**public** **void** printNo(**int**[] a, **int** n)

{

**for**(**int** i=0;i<n;i++)

System.***out***.println(a[i]);

}

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

BubbleSort ss=**new** BubbleSort();

// **TODO** Auto-generated method stub

Scanner sc=**new** Scanner(System.***in***);

System.***out***.println("How many no you want to Sort");

**int** n=sc.nextInt();

**int** a[]=**new** **int**[n];

**for**(**int** i=0;i<n;i++)

{

**int** x=sc.nextInt();

a[i]=x;

}

**int** b[]=ss.bubbleSort(a,n);

ss.printNo(b,n);

}

}

## Recursive Bubble Sort

**How to implement it recursively?**  
Recursive Bubble Sort has no performance/implementation advantages, but can be a good question to check one’s understanding of Bubble Sort and recursion.

If we take a closer look at Bubble Sort algorithm, we can notice that in first pass, we move largest element to end (Assuming sorting in increasing order). In second pass, we move second largest element to second last position and so on.

**Recursion Idea.**

1. Base Case: If array size is 1, return.
2. Do One Pass of normal Bubble Sort. This pass fixes last element of current subarray.
3. Recur for all elements except last of current subarray.

Below is implementation of above idea.

|  |
| --- |
| // Java program for recursive implementation  // of Bubble sort    import java.util.Arrays;    public class GFG  {      // A function to implement bubble sort      static void bubbleSort(int arr[], int n)      {          // Base case          if (n == 1)              return;            // One pass of bubble sort. After          // this pass, the largest element          // is moved (or bubbled) to end.          for (int i=0; i<n-1; i++)              if (arr[i] > arr[i+1])              {                  // swap arr[i], arr[i+1]                  int temp = arr[i];                  arr[i] = arr[i+1];                  arr[i+1] = temp;              }            // Largest element is fixed,          // recur for remaining array          bubbleSort(arr, n-1);      }        // Driver Method      public static void main(String[] args)      {          int arr[] = {64, 34, 25, 12, 22, 11, 90};            bubbleSort(arr, arr.length);            System.out.println("Sorted array : ");          System.out.println(Arrays.toString(arr));      }  } |

**Output :**

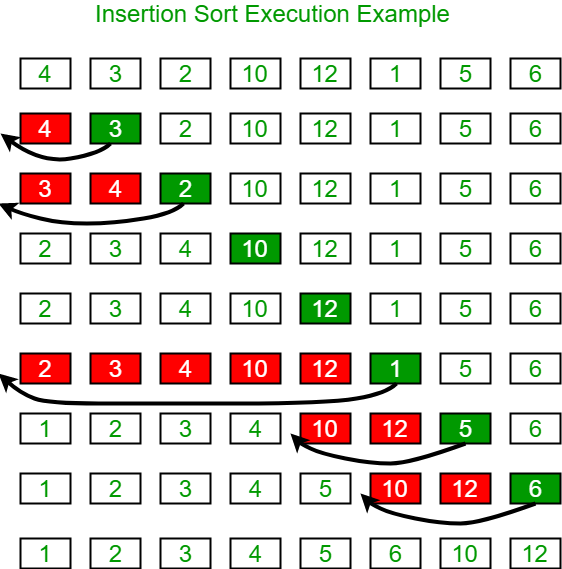
Sorted array :

11 12 22 25 34 64 90

## Insertion Sort

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

**Algorithm**  
// Sort an arr[] of size n  
insertionSort(arr, n)  
Loop from i = 1 to n-1.  
……a) Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

**Example:**  
[](https://media.geeksforgeeks.org/wp-content/uploads/insertionsort.png)  
**Another Example:**  
**12**, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 4 (last element of the array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12  
**11, 12**, 13, 5, 6

i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13  
**11, 12, 13**, 5, 6

i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.  
**5, 11, 12, 13**, 6

i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.  
**5, 6, 11, 12, 13**

## [Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/insertion-sort/1)

|  |
| --- |
| // Java program for implementation of Insertion Sort  class InsertionSort {      /\*Function to sort array using insertion sort\*/      void sort(int arr[])      {          int n = arr.length;          for (int i = 1; i < n; ++i) {              int key = arr[i];              int j = i - 1;                /\* Move elements of arr[0..i-1], that are                 greater than key, to one position ahead                 of their current position \*/              while (j >= 0 && arr[j] > key) {                  arr[j + 1] = arr[j];                  j = j - 1;              }              arr[j + 1] = key;          }      }        /\* A utility function to print array of size n\*/      static void printArray(int arr[])      {          int n = arr.length;          for (int i = 0; i < n; ++i)              System.out.print(arr[i] + " ");            System.out.println();      }        // Driver method      public static void main(String args[])      {          int arr[] = { 12, 11, 13, 5, 6 };            InsertionSort ob = new InsertionSort();          ob.sort(arr);            printArray(arr);      }  } /\* This code is contributed by Rajat Mishra. \*/ |

**Output:**

5 6 11 12 13

**Time Complexity:** O(n\*2)

**Auxiliary Space:**O(1)

**Boundary Cases**: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Using for loop

**import** java.util.Scanner;

**public** **class** InsertionSort {

**public** **int**[] insertionSort(**int**[] a,**int** n)

{

**for**(**int** i=1;i<n;i++)

{

**int** key=a[i];

**for**(**int** j=i-1;j>=0;j--)

{

**if**(a[j]>=key)

{

a[j+1]=a[j];

}

**else**

{a[j+1]=key;

**break**;}

}

}

**return** a;

}

**public** **void** printNo(**int**[] a, **int** n)

{

**for**(**int** i=0;i<n;i++)

System.***out***.println(a[i]);

}

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

InsertionSort ss=**new** InsertionSort();

// **TODO** Auto-generated method stub

Scanner sc=**new** Scanner(System.***in***);

System.***out***.println("How many no you want to Sort");

**int** n=sc.nextInt();

**int** a[]=**new** **int**[n];

**for**(**int** i=0;i<n;i++)

{

**int** x=sc.nextInt();

a[i]=x;

}

**int** b[]=ss.insertionSort(a,n);

ss.printNo(b,n);

}

}

# SQL

## [SQL](https://www.codecademy.com/learn/learn-sql) : [Queries](https://www.codecademy.com/learn/learn-sql/modules/learn-sql-queries)

**ANDOperator**

The AND operator allows multiple conditions to be combined. Records must match both conditions that are joined by AND to be included in the result set. The example query will match any car that is blue and made after 2014.

SELECT model FROM cars WHERE color = 'blue' AND year > 2014;

**ASClause**

Columns or tables in SQL can be *aliased* using the AS clause. This allows columns or tables to be specifically renamed in the returned result set. The given query will return a result set with the column for name renamed to movie\_title.

SELECT name AS 'movie\_title' FROM movies;

**OROperator**

The OR operator allows multiple conditions to be combined. Records matching either condition joined by the OR are included in the result set. The given query will match customers whose state is either ca or ny.

SELECT name FROM customers WHERE state = "ca" OR state = "ny";

**%Wildcard**

The % wildcard can be used in a LIKE operator pattern to match zero or more unspecified character(s). The example query will match any movie that begins with The, followed by zero or more of any characters.

SELECT name FROM movies WHERE name LIKE 'The%';

**SELECTStatement**

The SELECT \* statement returns all columns from the provided table in the result set. The given query will fetch all columns and records (rows) from the movies table.

SELECT \* FROM movies;

**\_Wildcard**

The \_ wildcard can be used in a LIKE operator pattern to match any single unspecified character. The given query will match any movie which begins with a single character, followed by ove.

SELECT name FROM movies WHERE name LIKE '\_ove';

**ORDER BYClause**

The ORDER BY clause can be used to sort the result set by a particular column either alphabetically or numerically. It can be ordered in ascending (default) or descending order with ASC/DESC. In the example, all the rows of the contacts table will be ordered by the birth\_date column in descending order.

SELECT \* FROM contacts ORDER BY birth\_date DESC;

**LIKEOperator**

The LIKE operator can be used inside of a WHERE clause to match a specified pattern. The given query will match any movie that begins with Star in its title.

SELECT name FROM movies WHERE name LIKE 'Star%';

**DISTINCTClause**

Unique values of a column can be selected using a DISTINCT query. For a table contact\_details having five rows in which the city column contains Chicago, Madison, Boston, Madison, and Denver, the given query would return:

* Chicago
* Madison
* Boston
* Denver

SELECT DISTINCT city FROM contact\_details;

**BETWEENOperator**

The BETWEEN operator can be used to filter by a *range* of values. The range of values can be text, numbers or date data. The given query will match any movie made between the years 1980 and 1990, inclusive.

SELECT \* FROM movies WHERE year BETWEEN 1980 AND 1990;

**LIMITClause**

The LIMIT clause is used to narrow, or *limit*, a result set to the specified number of rows. The given query will limit the result set to 5 rows.

SELECT \* FROM movies LIMIT 5;

**NULLValues**

Column values in SQL records can be NULL, or have no value. These records can be matched (or not matched) using the IS NULL and IS NOT NULL operators in combination with the WHERE clause. The given query will match all addresses where the address has a value or is not NULL.

SELECT address FROM records WHERE address IS NOT NULL;

**WHEREClause**

The WHERE clause is used to filter records (rows) that match a certain condition. The given query will select all records where the pub\_year equals 2017.

SELECT title FROM library WHERE pub\_year = 2017;

## [SQL](https://www.codecademy.com/learn/learn-sql) : [Manipulation](https://www.codecademy.com/learn/learn-sql/modules/learn-sql-manipulation)

[**Download as a PDF**](https://codecademy-content.s3.amazonaws.com/cheatsheets/learn-sql/learn-sql-manipulation.pdf)

**Column Constraints**

Column constraints are the rules applied to the values of individual columns:

* PRIMARY KEY constraint can be used to uniquely identify the row.
* UNIQUE columns have a different value for every row.
* NOT NULL columns must have a value.
* DEFAULT assigns a default value for the column when no value is specified.

There can be only one PRIMARY KEY column per table and multiple UNIQUE columns.

CREATE TABLE student ( id INTEGER PRIMARY KEY, name TEXT UNIQUE, grade INTEGER NOT NULL, age INTEGER DEFAULT 10 );

**CREATE TABLEStatement**

The CREATE TABLE statement is used to create a new table in a database. It allows one to specify the name of the table and the name of each column in the table.

CREATE TABLE table\_name ( column1 datatype, column2 datatype, column3 datatype );

**INSERTStatement**

The INSERT INTO statement is used to add a new record (row) to a table.

It has two forms as shown in the code block:

* Insert values based on the order of the columns in the table.
* Define the columns to insert values into.

-- Insert into columns in order: INSERT INTO table\_name VALUES (value1, value2, value3); -- Insert into columns by name: INSERT INTO table\_name (column1, column2, column3) VALUES (value1, value2, value3);

**ALTER TABLEStatement**

The ALTER TABLE statement is used to modify the columns of an existing table. When combined with the ADD COLUMN clause, it is used to add a new column to a table.

-- Syntax: ALTER TABLE table\_name ADD column\_name datatype; -- Example: ALTER TABLE employees ADD first\_name TEXT;

**DELETEStatement**

The DELETE statement is used to delete records (rows) in a table. This statement does not delete the whole table.

Inside, the WHERE clause specifies which record or records that should be deleted. If the WHERE clause is omitted, all records will be deleted.

DELETE FROM table\_name WHERE some\_column = some\_value;

**UPDATEStatement**

The UPDATE statement is used to edit records (rows) in a table. It usually includes a SET clause that indicates the column to edit and a WHERE clause for specifying which record(s) should be updated.

UPDATE table\_name SET column1 = value1, column2 = value2 WHERE some\_column = some\_value;

## [SQL](https://www.codecademy.com/learn/learn-sql) : [Aggregate Functions](https://www.codecademy.com/learn/learn-sql/modules/learn-sql-aggregate-functions)

[**Download as a PDF**](https://codecademy-content.s3.amazonaws.com/cheatsheets/learn-sql/learn-sql-aggregate-functions.pdf)

**Column References**

The GROUP BY and ORDER BY clauses can reference the selected columns by number in which they appear in the SELECT statement. The example query will count the number of movies per rating, and will:

* GROUP BY column 2 (rating)
* ORDER BY column 1 (total\_movies)

SELECT COUNT(\*) AS 'total\_movies', rating FROM movies GROUP BY 2 ORDER BY 1;

**SUM()Aggregate Function**

The SUM() aggregate function takes the name of a column as an argument and returns the sum of all the value in that column.

SELECT SUM(salary) FROM salary\_disbursement;

**MAX()Aggregate Function**

The MAX() aggregate function in SQL takes the name of a column as an argument and returns the largest value in a column. The given query will return the largest value from the amount column.

SELECT MAX(amount) FROM transactions;

**COUNT()Aggregate Function**

The COUNT() aggregate function in SQL returns the total number of rows that match the specified criteria. For instance, to find the total number of employees who have less than 5 years of experience, the given query can be used.

**Note:** A column name of the table can also be used instead of \*. Unlike COUNT(\*), this variation COUNT(column) will not count NULL values in that column.

SELECT COUNT(\*) FROM employees WHERE experience < 5;

**GROUP BYClause**

The GROUP BY clause will group records in a result set by identical values in one or more columns. It is often used in combination with aggregate functions to query information of similar records. The GROUP BY clause can come after FROM or WHERE but must come before any ORDER BY or LIMIT clause.

The given query will count the number of movies per rating.

SELECT rating, COUNT(\*) FROM movies GROUP BY rating;

**MIN()Aggregate Function**

The MIN() aggregate function in SQL returns the smallest value in a column. For instance, to find the smallest value of the amount column from the table named transactions, the given query can be used.

SELECT MIN(amount) FROM transactions;

**AVG()Aggregate Function**

The AVG() aggregate function returns the average value in a column. For instance, to find the average salary for the employees who have less than 5 years of experience, the given query can be used.

SELECT AVG(salary) FROM employees WHERE experience < 5;

**HAVINGClause**

The HAVING clause is used to further filter the result set groups provided by the GROUP BY clause. HAVING is often used with aggregate functions to filter the result set groups based on an aggregate property. The given query will select only the records (rows) from only years where more than 5 movies were released per year.

SELECT year, COUNT(\*) FROM movies GROUP BY year HAVING COUNT(\*) > 5;

**ROUND()Function**

The ROUND() function will round a number value to a specified number of places. It takes two arguments: a number, and a number of decimal places. It can be combined with other aggregate functions, as shown in the given query. This query will calculate the average rating of movies from 2015, rounding to 2 decimal places.

SELECT year, ROUND(AVG(rating), 2) FROM movies WHERE year = 2015;

## [SQL](https://www.codecademy.com/learn/learn-sql) : [Multiple Tables](https://www.codecademy.com/learn/learn-sql/modules/learn-sql-multiple-tables)

[**Download as a PDF**](https://codecademy-content.s3.amazonaws.com/cheatsheets/learn-sql/learn-sql-multiple-tables.pdf)

**Outer Join**

An outer join will combine rows from different tables even if the join condition is not met. In a LEFT JOIN, every row in the left table is returned in the result set, and if the join condition is not met, then NULL values are used to fill in the columns from the right table.

SELECT column\_name(s) FROM table1 LEFT JOIN table2 ON table1.column\_name = table2.column\_name;

**WITHClause**

The WITH clause stores the result of a query in a temporary table (temporary\_movies) using an alias.

Multiple temporary tables can be defined with one instance of the WITH keyword.

WITH temporary\_movies AS ( SELECT \* FROM movies ) SELECT \* FROM temporary\_movies WHERE year BETWEEN 2000 AND 2020;

**UNIONClause**

The UNION clause is used to combine results that appear from multiple SELECT statements and filter duplicates.

For example, given a first\_names table with a column name containing rows of data “James” and “Hermione”, and a last\_names table with a column name containing rows of data “James”, “Hermione” and “Cassidy”, the result of this query would contain three names: “Cassidy”, “James”, and “Hermione”.

SELECT name FROM first\_names UNION SELECT name FROM last\_names

**CROSS JOINClause**

The CROSS JOIN clause is used to combine each row from one table with each row from another in the result set. This JOIN is helpful for creating all possible combinations for the records (rows) in two tables.

The given query will select the shirt\_color and pants\_color columns from the result set, which will contain all combinations of combining the rows in the shirts and pants tables. If there are 3 different shirt colors in the shirts table and 5 different pants colors in the pants table then the result set will contain 3 x 5 = 15 rows.

SELECT shirts.shirt\_color, pants.pants\_color FROM shirts CROSS JOIN pants;

**Inner Join**

The JOIN clause allows for the return of results from more than one table by joining them together with other results based on common column values specified using an ON clause. INNER JOIN is the default JOIN and it will only return results matching the condition specified by ON.

SELECT \* FROM books JOIN authors ON books.author\_id = authors.id;

## COMMANDS

### ALTER TABLE

ALTER TABLE table\_name

ADD column\_name datatype;

ALTER TABLE lets you add columns to a table in a database.

### AND

SELECT column\_name(s)

FROM table\_name

WHERE column\_1 = value\_1

AND column\_2 = value\_2;

AND is an operator that combines two conditions. Both conditions must be true for the row to be included in the result set.

### AS

SELECT column\_name AS 'Alias'

FROM table\_name;

AS is a keyword in SQL that allows you to rename a column or table using an alias.

### AVG()

SELECT AVG(column\_name)

FROM table\_name;

AVG() is an aggregate function that returns the average value for a numeric column.

### BETWEEN

SELECT column\_name(s)

FROM table\_name

WHERE column\_name BETWEEN value\_1 AND value\_2;

The BETWEEN operator is used to filter the result set within a certain range. The values can be numbers, text or dates.

### CASE

SELECT column\_name,

CASE

WHEN condition THEN 'Result\_1'

WHEN condition THEN 'Result\_2'

ELSE 'Result\_3'

END

FROM table\_name;

CASE statements are used to create different outputs (usually in the SELECT statement). It is SQL’s way of handling if-then logic.

### COUNT()

SELECT COUNT(column\_name)

FROM table\_name;

COUNT() is a function that takes the name of a column as an argument and counts the number of rows where the column is not NULL.

### CREATE TABLE

CREATE TABLE table\_name (

column\_1 datatype,

column\_2 datatype,

column\_3 datatype

);

CREATE TABLE creates a new table in the database. It allows you to specify the name of the table and the name of each column in the table.

### DELETE

DELETE FROM table\_name

WHERE some\_column = some\_value;

DELETE statements are used to remove rows from a table.

### GROUP BY

SELECT column\_name, COUNT(\*)

FROM table\_name

GROUP BY column\_name;

GROUP BY is a clause in SQL that is only used with aggregate functions. It is used in collaboration with the SELECT statement to arrange identical data into groups.

### HAVING

SELECT column\_name, COUNT(\*)

FROM table\_name

GROUP BY column\_name

HAVING COUNT(\*) > value;

HAVING was added to SQL because the WHERE keyword could not be used with **aggregate functions**.

### INNER JOIN

SELECT column\_name(s)

FROM table\_1

JOIN table\_2

ON table\_1.column\_name = table\_2.column\_name;

An inner join will combine rows from different tables if the join condition is true.

### INSERT

INSERT INTO **table\_name (column\_1, column\_2, column\_3)**

**VALUES (value\_1, 'value\_2', value\_3);**

INSERT statements are used to add a new row to a table.

### IS NULL / IS NOT NULL

SELECT column\_name(s)

FROM table\_name

WHERE column\_name IS NULL;

IS NULL and IS NOT NULL are operators used with the WHERE clause to test for empty values.

### LIKE

SELECT column\_name(s)

FROM table\_name

WHERE column\_name LIKE pattern;

LIKE is a special operator used with the WHERE clause to search for a specific pattern in a column.

### LIMIT

SELECT column\_name(s)

FROM table\_name

LIMIT number;

LIMIT is a clause that lets you specify the maximum number of rows the result set will have.

### MAX()

SELECT MAX(column\_name)

FROM table\_name;

MAX() is a function that takes the name of a column as an argument and returns the largest value in that column.

### MIN()

SELECT MIN(column\_name)

FROM table\_name;

MIN() is a function that takes the name of a column as an argument and returns the smallest value in that column.

### OR

SELECT column\_name

FROM table\_name

WHERE column\_name = value\_1

OR column\_name = value\_2;

OR is an operator that filters the result set to only include rows where either condition is true.

### ORDER BY

SELECT column\_name

FROM table\_name

ORDER BY column\_name ASC | DESC;

ORDER BY is a clause that indicates you want to sort the result set by a particular column either alphabetically or numerically.

### OUTER JOIN

SELECT column\_name(s)

FROM table\_1

LEFT JOIN table\_2

ON table\_1.column\_name = table\_2.column\_name;

An outer join will combine rows from different tables even if the join condition is not met. Every row in the left table is returned in the result set, and if the join condition is not met, then NULL values are used to fill in the columns from the right table.

### ROUND()

SELECT ROUND(column\_name, integer)

FROM table\_name;

ROUND() is a function that takes a column name and an integer as arguments. It rounds the values in the column to the number of decimal places specified by the integer.

### SELECT

SELECT column\_name

FROM table\_name;

SELECT statements are used to fetch data from a database. Every query will begin with SELECT.

### SELECT DISTINCT

SELECT DISTINCT column\_name

FROM table\_name;

SELECT DISTINCT specifies that the statement is going to be a query that returns unique values in the specified column(s).

### SUM

SELECT SUM(column\_name)

FROM table\_name;

SUM() is a function that takes the name of a column as an argument and returns the sum of all the values in that column.

### UPDATE

UPDATE table\_name

SET some\_column = some\_value

WHERE some\_column = some\_value;

UPDATE statements allow you to edit rows in a table.

### WHERE

SELECT column\_name(s)

FROM table\_name

WHERE column\_name operator value;

WHERE is a clause that indicates you want to filter the result set to include only rows where the following condition is true.

### WITH

WITH temporary\_name AS (

SELECT \*

FROM table\_name)

SELECT \*

FROM temporary\_name

WHERE column\_name operator value;

WITH clause lets you store the result of a query in a temporary table using an alias. You can also define multiple temporary tables using a comma and with one instance of the WITH keyword.

The WITH clause is also known as common table expression (CTE) and subquery factoring.

## Top 50 SQL Interview Questions & Answers

**1. What is DBMS?**

A Database Management System (DBMS) is a program that controls creation, maintenance and use of a database. DBMS can be termed as File Manager that manages data in a database rather than saving it in file systems.

**2. What is RDBMS?**

RDBMS stands for Relational Database Management System. RDBMS store the data into the collection of tables, which is related by common fields between the columns of the table. It also provides relational operators to manipulate the data stored into the tables.

**Example: SQL Server.**

**3. What is SQL?**

SQL stands for Structured Query Language , and it is used to communicate with the Database. This is a standard language used to perform tasks such as retrieval, updation, insertion and deletion of data from a database.

Standard SQL Commands are Select.

**4. What is a Database?**

Database is nothing but an organized form of data for easy access, storing, retrieval and managing of data. This is also known as structured form of data which can be accessed in many ways.

Example: School Management Database, Bank Management Database.

**5. What are tables and Fields?**

A table is a set of data that are organized in a model with Columns and Rows. Columns can be categorized as vertical, and Rows are horizontal. A table has specified number of column called fields but can have any number of rows which is called record.

Example:.

Table: Employee.

Field: Emp ID, Emp Name, Date of Birth.

Data: 201456, David, 11/15/1960.

**6. What is a primary key?**

A primary key is a combination of fields which uniquely specify a row. This is a special kind of unique key, and it has implicit NOT NULL constraint. It means, Primary key values cannot be NULL.

**7. What is a unique key?**

A Unique key constraint uniquely identified each record in the database. This provides uniqueness for the column or set of columns.

A Primary key constraint has automatic unique constraint defined on it. But not, in the case of Unique Key.

There can be many unique constraint defined per table, but only one Primary key constraint defined per table.

**8. What is a foreign key?**

A foreign key is one table which can be related to the primary key of another table. Relationship needs to be created between two tables by referencing foreign key with the primary key of another table.

**9. What is a join?**

This is a keyword used to query data from more tables based on the relationship between the fields of the tables. Keys play a major role when JOINs are used.

**10. What are the types of join and explain each?**

There are various types of join which can be used to retrieve data and it depends on the relationship between tables.

* **Inner Join.**

Inner join return rows when there is at least one match of rows between the tables.

* **Right Join.**

Right join return rows which are common between the tables and all rows of Right hand side table. Simply, it returns all the rows from the right hand side table even though there are no matches in the left hand side table.

* **Left Join.**

Left join return rows which are common between the tables and all rows of Left hand side table. Simply, it returns all the rows from Left hand side table even though there are no matches in the Right hand side table.

* **Full Join.**

Full join return rows when there are matching rows in any one of the tables. This means, it returns all the rows from the left hand side table and all the rows from the right hand side table.

**11. What is normalization?**

Normalization is the process of minimizing redundancy and dependency by organizing fields and table of a database. The main aim of Normalization is to add, delete or modify field that can be made in a single table.

**12. What is Denormalization.**

DeNormalization is a technique used to access the data from higher to lower normal forms of database. It is also process of introducing redundancy into a table by incorporating data from the related tables.

**13. What are all the different normalizations?**

The normal forms can be divided into 5 forms, and they are explained below -.

* **First Normal Form (1NF):.**

This should remove all the duplicate columns from the table. Creation of tables for the related data and identification of unique columns.

* **Second Normal Form (2NF):.**

Meeting all requirements of the first normal form. Placing the subsets of data in separate tables and Creation of relationships between the tables using primary keys.

* **Third Normal Form (3NF):.**

This should meet all requirements of 2NF. Removing the columns which are not dependent on primary key constraints.

* **Fourth Normal Form (4NF):.**

Meeting all the requirements of third normal form and it should not have multi- valued dependencies.

**14. What is a View?**

A view is a virtual table which consists of a subset of data contained in a table. Views are not virtually present, and it takes less space to store. View can have data of one or more tables combined, and it is depending on the relationship.

**15. What is an Index?**

An index is performance tuning method of allowing faster retrieval of records from the table. An index creates an entry for each value and it will be faster to retrieve data.

**16. What are all the different types of indexes?**

There are three types of indexes -.

* **Unique Index.**

This indexing does not allow the field to have duplicate values if the column is unique indexed. Unique index can be applied automatically when primary key is defined.

* **Clustered Index.**

This type of index reorders the physical order of the table and search based on the key values. Each table can have only one clustered index.

* **NonClustered Index.**

NonClustered Index does not alter the physical order of the table and maintains logical order of data. Each table can have 999 nonclustered indexes.

**17. What is a Cursor?**

A database Cursor is a control which enables traversal over the rows or records in the table. This can be viewed as a pointer to one row in a set of rows. Cursor is very much useful for traversing such as retrieval, addition and removal of database records.

**18. What is a relationship and what are they?**

Database Relationship is defined as the connection between the tables in a database. There are various data basing relationships, and they are as follows:.

* One to One Relationship.
* One to Many Relationship.
* Many to One Relationship.
* Self-Referencing Relationship.

**19. What is a query?**

A DB query is a code written in order to get the information back from the database. Query can be designed in such a way that it matched with our expectation of the result set. Simply, a question to the Database.

**20. What is subquery?**

A subquery is a query within another query. The outer query is called as main query, and inner query is called subquery. SubQuery is always executed first, and the result of subquery is passed on to the main query.

**21. What are the types of subquery?**

There are two types of subquery – Correlated and Non-Correlated.

A correlated subquery cannot be considered as independent query, but it can refer the column in a table listed in the FROM the list of the main query.

A Non-Correlated sub query can be considered as independent query and the output of subquery are substituted in the main query.

**22. What is a stored procedure?**

Stored Procedure is a function consists of many SQL statement to access the database system. Several SQL statements are consolidated into a stored procedure and execute them whenever and wherever required.

**23. What is a trigger?**

A DB trigger is a code or programs that automatically execute with response to some event on a table or view in a database. Mainly, trigger helps to maintain the integrity of the database.

Example: When a new student is added to the student database, new records should be created in the related tables like Exam, Score and Attendance tables.

**24. What is the difference between DELETE and TRUNCATE commands?**

DELETE command is used to remove rows from the table, and WHERE clause can be used for conditional set of parameters. Commit and Rollback can be performed after delete statement.

TRUNCATE removes all rows from the table. Truncate operation cannot be rolled back.

**25. What are local and global variables and their differences?**

Local variables are the variables which can be used or exist inside the function. They are not known to the other functions and those variables cannot be referred or used. Variables can be created whenever that function is called.

Global variables are the variables which can be used or exist throughout the program. Same variable declared in global cannot be used in functions. Global variables cannot be created whenever that function is called.

**26. What is a constraint?**

Constraint can be used to specify the limit on the data type of table. Constraint can be specified while creating or altering the table statement. Sample of constraint are.

* NOT NULL.
* CHECK.
* DEFAULT.
* UNIQUE.
* PRIMARY KEY.
* FOREIGN KEY.

**27. What is data Integrity?**

Data Integrity defines the accuracy and consistency of data stored in a database. It can also define integrity constraints to enforce business rules on the data when it is entered into the application or database.

**28. What is Auto Increment?**

Auto increment keyword allows the user to create a unique number to be generated when a new record is inserted into the table. AUTO INCREMENT keyword can be used in Oracle and IDENTITY keyword can be used in SQL SERVER.

Mostly this keyword can be used whenever PRIMARY KEY is used.

**29. What is the difference between Cluster and Non-Cluster Index?**

Clustered index is used for easy retrieval of data from the database by altering the way that the records are stored. Database sorts out rows by the column which is set to be clustered index.

A nonclustered index does not alter the way it was stored but creates a complete separate object within the table. It point back to the original table rows after searching.

**30. What is Datawarehouse?**

Datawarehouse is a central repository of data from multiple sources of information. Those data are consolidated, transformed and made available for the mining and online processing. Warehouse data have a subset of data called Data Marts.

# Node.js | NPM (Node Package Manager(just like maven..ie it is maven of js))

**NPM (Node Package Manager)** is the default package manager for Node.js and is written entirely in [Javascript](https://www.geeksforgeeks.org/javascript-tutorial/). Developed by Isaac Z. Schlueter, it was initially released in January 12, 2010. NPM manages all the packages and modules for Node.js and consists of command line client **npm**. It gets installed into the system with installation of Node.js. The required packages and modules in Node project are installed using NPM.  
A package contains all the files needed for a module and modules are the JavaScript libraries that can be included in Node project according to the requirement of the project.  
NPM can install all the dependencies of a project through the [package.json](https://www.geeksforgeeks.org/node-js-package-json/) file. It can also update and uninstall packages. In the [package.json](https://www.geeksforgeeks.org/node-js-package-json/) file, each dependency can specify a range of valid versions using the semantic versioning scheme, allowing developers to auto-update their packages while at the same time avoiding unwanted breaking changes.

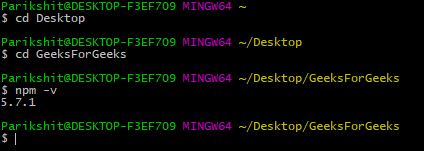
**Some facts about NPM:**

* At the time of writing this article, NPM has 580096 registered packages. The average rate of growth of this number is 291/day which outraces every other package registry.
* npm is open source
* The top npm packages in the decreasing order are: lodash, async, react, request, express.

**Installing NPM:**  
To install NPM, it is required to install Node.js as NPM gets installed with Node.js automatically.  
[Install Node.js](https://nodejs.org/en/).

**Checking and updating npm version:**  
Version of **npm** installed on system can be checked using following syntax:  
**Syntax:**

**npm -v**



*Checking npm version*

If the installed version is not latest, one can always update it using the given syntax:  
**Syntax:**

**npm npm@latest -g**.

As **npm** is a global package, **-g** flag is used to update it **globally**.

**Creating a Node Project:**  
To create a Node project, **npm init** is used in the folder in which user want to create project. The npm command line will ask a number of questions like **name, license, scripts, description, author, keywords, version, main file** etc. After npm is done creating the project, a [package.json](https://www.geeksforgeeks.org/node-js-package-json/) file will be visible in project folder as a proof that the project has been initialized.



*npm init*

**Installing Packages:**  
After creating the project, next step is to incorporate the packages and modules to be used in the Node Project. To install packages and modules in the project use the following syntax:  
**Syntax:**

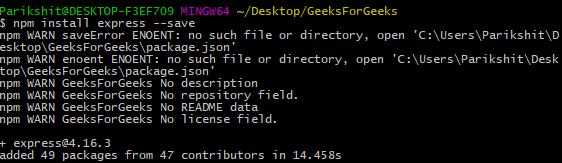
**npm install package\_name**

**Example:** Installing the express package into the project. Express is the web development framework used by the Node.  
**Syntax:**

**npm install express**

To use express in the Node, follow the below syntax:  
**Syntax:**

var express = require('express');



*Installing express module*

**Example:** To install a package globally (accessible by all projects in system), add an extra **-g** tag in syntax used to install the package.  
Installing **nodemon** package globally.

**npm install nodemon -g**



*Installing nodemon package globally*

**Controlling where the package gets installed:**  
To install a package and simultaneously save it in [package.json](https://www.geeksforgeeks.org/node-js-package-json/) file (in case using Node.js), add **–save** flag. The **–save** flag is default in npm install command so it is equal to **npm install package\_name** command.  
**Example:**

**npm install express --save**

By **–save** flag one can control where the packages are to be installed.  
**–save-prod :** Using this packages will appear in Dependencies which is also by default.  
**–save-dev :** Using this packages will get appear in devDependencies and will only be used in the development mode.  
**Example:** npm install node-color –save-dev



*–save-dev example*

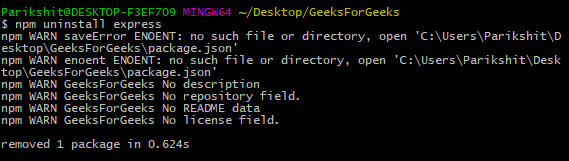
If there is a [package.json](https://www.geeksforgeeks.org/node-js-package-json/) file with all the packages mentioned as dependencies already, just type **npm install** in terminal. **npm** will look at package.json file and install all the dependencies according to their mentioned versions. This command is typically used when a Node project is forked and cloned. The **node\_modules** being a big folder is generally not pushed to a github repo and the cloner has to run **npm install** to install the dependencies.

**Note:** NPM installs the dependencies in local mode (Default) which go to the **node\_modules** directory present in the folder of Node application. To see all the locally installed modules use **npm ls** command.

**Uninstalling Packages:**  
To uninstall packages using npm, follow the below syntax:  
**Syntax:**

**npm uninstall**

**Example:** To uninstall the express package



*Uninstalling express*

To uninstall global packages, follow the below syntax:  
**Syntax:**

**npm uninstall package\_name -g**

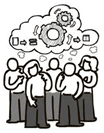
**Using Semantic Versioning to manage packages:**  
versioning major minor patch explanation

* To install a package of a specific version, mention the full and exact version in the [package.json](https://www.geeksforgeeks.org/node-js-package-json/) file.
* To install the latest version of the package, mention “\*” in front of the dependency or “latest”. This will find the latest stable version of the module and install it.
* To install any version (stable one) above a given version, mention it like in the example below:  
  **“express”:”^4.1.1″.** in [package.json](https://www.geeksforgeeks.org/node-js-package-json/) file. The caret symbol (^) is used to tell the npm to find a version greater than 4.1.1 and install it.

# Agile Methodology & Model: Guide for Software Development & Testing

## What is Agile Methodology?

AGILE methodology is a practice that promotes **continuous iteration** of development and testing throughout the software development lifecycle of the project. Both development and testing activities are concurrent unlike the Waterfall model

[](https://www.guru99.com/images/11-2014/agile_Processesv1_1.png)

The agile software development emphasizes on four core values.

1. Individual and team interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

In this Software Engineering tutorial, you will learn

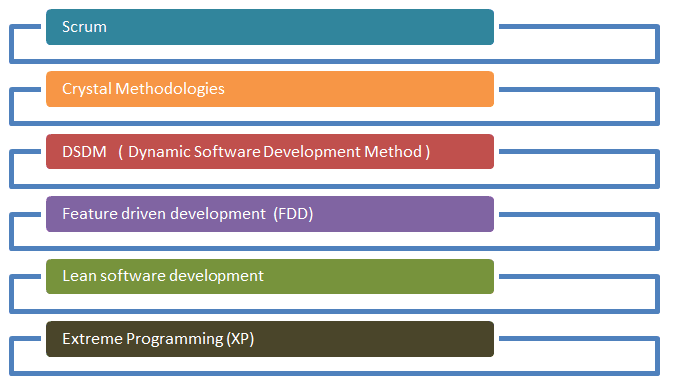
* [What is Agile Methodology?](https://www.guru99.com/agile-scrum-extreme-testing.html#1)
* [Agile Vs Waterfall Method](https://www.guru99.com/agile-scrum-extreme-testing.html#2)
* [Scrum](https://www.guru99.com/agile-scrum-extreme-testing.html#3)
* [Product Backlog](https://www.guru99.com/agile-scrum-extreme-testing.html#4)
* [Scrum Practices](https://www.guru99.com/agile-scrum-extreme-testing.html#5)
* [Process flow of Scrum Methodologies:](https://www.guru99.com/agile-scrum-extreme-testing.html#6)
* [eXtreme Programming (XP)](https://www.guru99.com/agile-scrum-extreme-testing.html#7)
* [Phases of eXtreme programming:](https://www.guru99.com/agile-scrum-extreme-testing.html#8)
* [Crystal Methodologies](https://www.guru99.com/agile-scrum-extreme-testing.html#9)
* [Dynamic Software Development Method (DSDM)](https://www.guru99.com/agile-scrum-extreme-testing.html#10)
* [Feature Driven Development (FDD)](https://www.guru99.com/agile-scrum-extreme-testing.html#11)
* [Lean Software Development](https://www.guru99.com/agile-scrum-extreme-testing.html#12)
* [Kanban](https://www.guru99.com/agile-scrum-extreme-testing.html#13)
* [Agile metrics](https://www.guru99.com/agile-scrum-extreme-testing.html#14)

## Agile Vs Waterfall Method

Agile and Waterfall model are two different methods for software development process. Though they are different in their approach, both methods are useful at times, depending on the requirement and the type of the project.

|  |  |
| --- | --- |
| **Agile Model** | **Waterfall Model** |
| * Agile method proposes incremental and iterative approach to software design | * Development of the software flows sequentially from start point to end point. |
| * The **agile process** is broken into individual models that designers work on | * The design process is not broken into an individual models |
| * The customer has early and frequent opportunities to look at the product and make decision and changes to the project | * The customer can only see the product at the end of the project |
| * Agile model is considered unstructured compared to the waterfall model | * Waterfall model are more secure because they are so plan oriented |
| * Small projects can be implemented very quickly. For large projects, it is difficult to estimate the development time. | * All sorts of project can be estimated and completed. |
| * Error can be fixed in the middle of the project. | * Only at the end, the whole product is tested. If the requirement error is found or any changes have to be made, the project has to start from the beginning |
| * Development process is iterative, and the project is executed in short (2-4) weeks iterations. Planning is very less. | * The development process is phased, and the phase is much bigger than iteration. Every phase ends with the detailed description of the next phase. |
| * Documentation attends less priority than software development | * Documentation is a top priority and can even use for training staff and upgrade the software with another team |
| * Every iteration has its own testing phase. It allows implementing regression testing every time new functions or logic are released. | * Only after the development phase, the testing phase is executed because separate parts are not fully functional. |
| * In agile testing when an iteration end, shippable features of the product is delivered to the customer. New features are usable right after shipment. It is useful when you have good contact with customers. | * All features developed are delivered at once after the long implementation phase. |
| * Testers and developers work together | * Testers work separately from developers |
| * At the end of every sprint, user acceptance is performed | * User acceptance is **performed** at the end of the project. |
| * It requires close communication with developers and together analyze requirements and planning | * Developer does not involve in requirement and planning process. Usually, time delays between tests and coding |

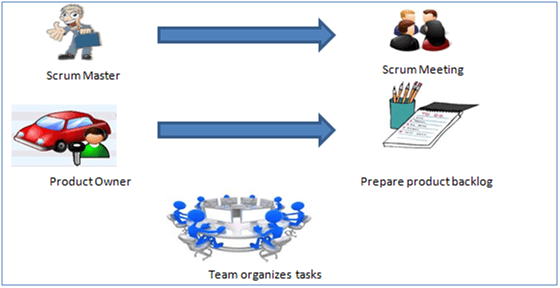
### Agile Methodology

[](https://www.guru99.com/images/11-2014/agile_Processesv1_2.png)

There are various **methods** present in agile testing, and those are listed below:

## Scrum

SCRUM is an agile development method which concentrates specifically on how to manage tasks within a team-based development environment. Basically, Scrum is derived from activity that occurs during a rugby match. Scrum believes in empowering the development team and advocates working in small teams (say- 7 to 9 members). It consists of three roles, and their responsibilities are explained as follows:

[](https://www.guru99.com/images/11-2014/agile_Processesv1_3.png)

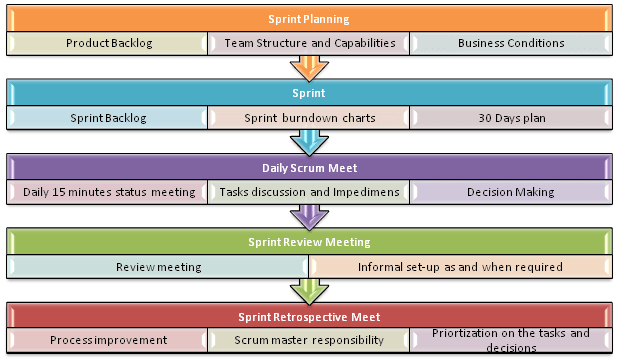
* Scrum Master
  + Master is responsible for setting up the team, sprint meeting and removes obstacles to progress
* Product owner
  + The Product Owner creates product backlog, prioritizes the backlog and is responsible for the delivery of the functionality at each iteration
* Scrum Team
  + Team manages its own work and organizes the work to complete the sprint or cycle

## Product Backlog

This is a repository where requirements are tracked with details on the no of requirements(user stories) to be completed for each release. It should be maintained and prioritized by Product Owner, and it should be distributed to the scrum team. Team can also request for a new requirement addition or modification or deletion

## Scrum Practices

Practices are described in detailed:

[](https://www.guru99.com/images/11-2014/agile_Processesv1_4.png)

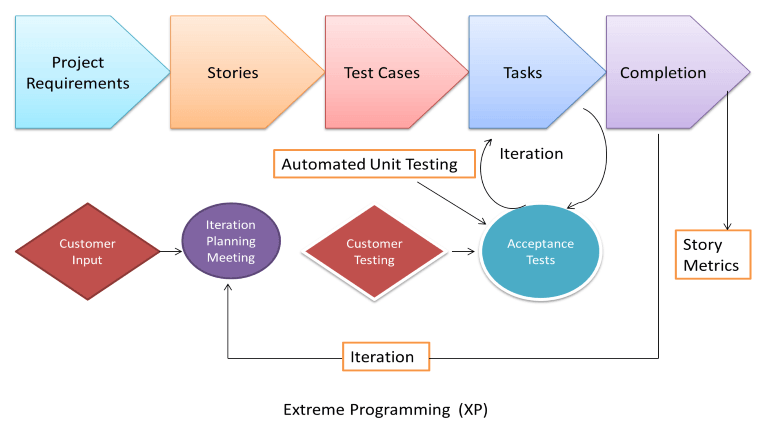
## Process flow of Scrum Methodologies:

Process flow of scrum testing is as follows:

* Each iteration of a scrum is known as Sprint
* Product backlog is a list where all details are entered to get the end-product
* During each Sprint, top user stories of Product backlog are selected and turned into Sprint backlog
* Team works on the defined sprint backlog
* Team checks for the daily work
* At the end of the sprint, team delivers product functionality

## eXtreme Programming (XP)

Extreme Programming technique is very helpful when there is constantly changing demands or requirements from the customers or when they are not sure about the functionality of the system. It advocates frequent "releases" of the product in short development cycles, which inherently improves the productivity of the system and also introduces a checkpoint where any customer requirements can be easily implemented. The XP develops software keeping customer in the target.

[](https://www.guru99.com/images/11-2014/agile_Processesv1_5.png)

Business requirements are gathered in terms of stories. All those stories are stored in a place called the parking lot.

In this type of methodology, releases are based on the shorter cycles called Iterations with span of 14 days time period. Each iteration includes phases like coding, unit testing and system testing where at each phase some minor or major functionality will be built in the application.

## Phases of eXtreme programming:

There are 6 phases available in Agile XP method, and those are explained as follows:

### Planning

* Identification of stakeholders and sponsors
* Infrastructure Requirements
* [Security](https://www.guru99.com/ethical-hacking-tutorials.html) related information and gathering
* Service Level Agreements and its conditions

### Analysis

* Capturing of Stories in Parking lot
* Prioritize stories in Parking lot
* Scrubbing of stories for estimation
* Define Iteration SPAN(Time)
* Resource planning for both Development and QA teams

### Design

* Break down of tasks
* Test Scenario preparation for each task
* Regression Automation Framework

### Execution

* Coding
* Unit Testing
* Execution of Manual test scenarios
* Defect Report generation
* Conversion of Manual to Automation regression test cases
* Mid Iteration review
* End of Iteration review

### Wrapping

* Small Releases
* Regression Testing
* Demos and reviews
* Develop new stories based on the need
* Process Improvements based on end of iteration review comments

### Closure

* Pilot Launch
* Training
* Production Launch
* SLA Guarantee assurance
* Review SOA strategy
* Production Support

There are two storyboards available to track the work on a daily basis, and those are listed below for reference.

* Story Cardboard
  + This is a traditional way of collecting all the stories in a board in the form of stick notes to track daily XP activities. As this manual activity involves more effort and time, it is better to switch to an online form.
* Online Storyboard
  + Online tool Storyboard can be used to store the stories. **Several teams can use it** for different purposes.

## Crystal Methodologies

Crystal Methodology is based on three concepts

1. **Chartering:** Various activities involved in this phase are creating a development team, performing a preliminary feasibility analysis, developing an initial plan and fine-tuning the development methodology
2. **Cyclic delivery:** The main development phase consists of two or more delivery cycles, during which the
   1. Team updates and refines the release plan
   2. Implements a subset of the requirements through one or more program test integrate iterations
   3. Integrated product is delivered to real users
   4. Review of the project plan and adopted development methodology
3. **Wrap Up:** The activities performed in this phase are deployment into the user environment, post- deployment reviews and reflections are performed.

## Dynamic Software Development Method (DSDM)

DSDM is a Rapid Application Development (RAD) approach to software development and provides an agile project delivery framework. The important aspect of DSDM is that the users are required to be involved actively, and the teams are given the power to make decisions. Frequent delivery of product becomes the active focus with DSDM. The techniques used in DSDM are

1. Time Boxing
2. MoSCoW Rules
3. Prototyping

The DSDM project consists of 7 phases

1. Pre-project
2. Feasibility Study
3. Business Study
4. Functional Model Iteration
5. Design and build Iteration
6. Implementation
7. Post-project

## Feature Driven Development (FDD)

This method is focused around "designing & building" features. Unlike other agile methods, FDD describes very specific and short phases of work that has to be accomplished separately per feature. It includes domain walkthrough, design inspection, promote to build, code inspection and design. FDD develops product keeping following things in the target

1. Domain object Modeling
2. Development by feature
3. Component/ Class Ownership
4. Feature Teams
5. Inspections
6. Configuration Management
7. Regular Builds
8. Visibility of progress and results

## Lean Software Development

Lean software development method is based on the principle "Just in time production". It aims at increasing speed of software development and decreasing cost. Lean development can be summarized in seven steps.

1. Eliminating Waste
2. Amplifying learning
3. Defer commitment (deciding as late as possible)
4. Early delivery
5. Empowering the team
6. Building Integrity
7. Optimize the whole

## Kanban

Kanban originally emerged from Japanese word that means, a card containing all the information needed to be done on the product at each stage along its path to completion. This framework or method is quite adopted in software testing method especially in agile testing.

### Scrum Vs Kanban

|  |  |
| --- | --- |
| **Scrum** | **Kanban** |
| * In scrum technique, test must be broken down so that they can be completed within one sprint | * No particular item size is prescribed |
| * Prescribes a prioritized product backlog | * Prioritization is optional |
| * Scrum team commits to a particular amount of work for the iteration | * Commitment is optional |
| * Burndown chart is prescribed | * No particular item size is prescribed |
| * Between each sprint, a scrum board is reset | * A Kanban board is persistent. It limits the number of items in workflow state |
| * It cannot add items to ongoing iteration | * It can add items whenever capacity is available |
| * WIP limited indirectly | * WIP limited directly |
| * Timeboxed iterations prescribed | * Timeboxed iterations optional |

## Agile metrics:

Metrics that can be collected for effective usage of Agile is:

* Drag Factor
  + Effort in hours which do not contribute to sprint goal
  + Drag factor can be improved by reducing number of shared resources, reducing the amount of non-contributing work
  + New estimates can be increased by percentage of drag factor -New estimate = (Old estimate+drag factor)
* Velocity
  + Amount of backlog(user stories) converted to shippable functionality of sprint
* No of Unit Tests added
* Time interval taken to complete daily build
* Bugs detected in an iteration or in previous iterations
* Production defect leakage

# Maven in Eclipse:

To create a maven project go to :

<https://start.spring.io/>

give data and generate the zip

import in eclipse as existing maven project

# Microservices

## Introduction

A **microservice** is a small, loosely coupled distributed service. Microservice Architectures evolved as a solution to the scalability and innovation challenges with Monolith architectures (Monolith applications are typically huge – more 100, 000 line of code). It allows you to take a large application and decompose or break into easily manageable small components with narrowly defined responsibilities.

**Reasons for using Microservice:**  
In monolith application, there are few challenges:

1. For a large application, it is difficult to understand the complexity and make code changes fast and correctly, sometimes it becomes hard to manage the code.
2. Applications need extensive manual testing to ensure the impact of changes.
3. For small change, the whole application needs to be built and deployed.
4. The heavy application slows down start-up time.

**Benefits of Microservices:**

1. **Small Modules –**  
   Application is broken into smaller modules which are easy for developers to code and maintain.
2. **Easier Process Adaption –**  
   By using microservices, new Technology & Process Adaption becomes easier. You can try new technologies with the newer microservices that we use.
3. **Independent scaling –**  
   Each microservice can scale independently via X-axis scaling (cloning with more CPU or memory) and Z-axis scaling (sharding), based upon their needs.
4. **Unaffected –**  
   Large applications remain largely unaffected by the failure of a single module.
5. **DURS –**  
   Each service can be independently DURS (deployed, updated, replaced, and scaled).

**Restrictions of Microservices:**

1. **Configuration Management –**  
   As it becomes granular the headache comes for configuring the services and monitoring those. You need to maintain configurations for hundreds of components across environments.
2. **Debugging –**  
   Tracking down the service failure is painstaking job. You might need to look into multiple services across different components. Centralized Logging and Dashboards are essential to make it easy to debug problems.
3. **Automation –** Because there are a number of smaller components instead of a monolith, you need to automate everything – Builds, Deployment, Monitoring etc.
4. **Testing –**  
   Needs a greater effort for end to end testing as it needs all the dependent services to be up and running.

**Microservice Frameworks for Java:**  
There are several microservices frameworks that you can use for developing for Java. Some of these are:

1. **Spring Boot –**  
   This is probably the best Java microservices framework that works on top of languages for Inversion of Control, Aspect Oriented Programming, and others.
2. **Dropwizard –**  
   Dropwizard pulls together stable, mature libraries from the Java ecosystem into a simple, light-weight package that lets you focus on getting things done.
3. **Restlet –**  
   Restlet Framework helps Java developers build better web APIs that follow the REST architecture style.
4. **Spark –**  
   A micro-framework for creating web applications in Kotlin and Java 8 with minimal effort.

Others that you can consider include: Ninja Web Framework, Play Framework, RestExpress and Restx Framework.

## Microservices with Spring Boot - Part 1 - Getting Started

*Jan 5, 2018*  
*9 minute read*

This guide will help you learn the basics of microservices and microservices architectures. We will also start looking at a basic implementation of a microservice with Spring Boot.

*We will create a couple of microservices and get them to talk to each other using Eureka Naming Server and Ribbon for Client Side Load Balancing.*

This is a 5 Part Article Series

Microservices with Spring Boot

* Current Part - Part 1 - Getting Started with Microservices Architecture
* Part 2 - [Creating Forex Microservice](https://www.springboottutorial.com/creating-microservices-with-spring-boot-part-2-forex-microservice)
* Part 3 - [Creating Currency Conversion Microservice](https://www.springboottutorial.com/creating-microservices-with-spring-boot-part-3-currency-conversion-microservice)
* Part 4 - [Using Ribbon for Load Balancing](https://www.springboottutorial.com/microservices-with-spring-boot-part-4-ribbon-for-load-balancing)
* Part 5 - [Using Eureka Naming Server](https://www.springboottutorial.com/microservices-with-spring-boot-part-5-eureka-naming-server)

In part 1 of this series, lets get introduced to the concept of microservices and understand how to create great microservices with Spring Boot and Spring Cloud.

### You will learn

* What is a Monolith?
* What is a Microservice?
* What are the Challenges with Microservices?
* How does Spring Boot and Spring Cloud make developing Microservices easy?
* How to implement client side load balancing with Ribbon?
* How to implement a Naming Server (Eureka Naming Server)?
* How to connect the microservices with Naming Server and Ribbon?

### 10 Step Reference Courses

* [Spring Framework for Beginners in 10 Steps](https://courses.in28minutes.com/p/spring-framework-for-beginners)
* [Spring Boot for Beginners in 10 Steps](https://courses.in28minutes.com/p/spring-boot-for-beginners-in-10-steps)
* [Spring MVC in 10 Steps](https://www.youtube.com/watch?v=BjNhGaZDr0Y)
* [JPA and Hibernate in 10 Steps](https://courses.in28minutes.com/p/jpa-and-hibernate-tutorial-for-beginners-with-spring-boot)
* [Eclipse Tutorial for Beginners in 5 Steps](https://courses.in28minutes.com/p/eclipse-tutorial-for-beginners)
* [Maven Tutorial for Beginners in 5 Steps](https://courses.in28minutes.com/p/maven-tutorial-for-beginners-in-5-steps)
* [JUnit Tutorial for Beginners in 5 Steps](https://courses.in28minutes.com/p/junit-tutorial-for-beginners)
* [Mockito Tutorial for Beginners in 5 Steps](https://courses.in28minutes.com/p/mockito-for-beginner-in-5-steps)
* [Complete in28Minutes Course Guide](https://courses.in28minutes.com/p/in28minutes-course-guide)

### Microservices Overview - A Big Picture

In this series of articles, we would create two microservices:

* Forex Service - Abbreviated as FS
* Currency Conversion Service - Abbreviated as CCS

*Do not worry if you are not clear about a few things. The idea is to give a big picture before we get our hands dirty and create the microservices step by step*

#### Forex Service

Forex Service (FS) is the Service Provider. It provides currency exchange values for various currency. Let’s assume that it talks to a Forex Exchange and provides the current conversion value between currencies.

An example request and response is shown below:

GET to http://localhost:8000/currency-exchange/from/EUR/to/INR

{

id: 10002,

from: "EUR",

to: "INR",

conversionMultiple: 75,

port: 8000,

}

The request above is the currency exchange value for EUR to INR. In the response, conversionMultiple is 75.

*We will talk about port in the response a little later.*

#### Currency Conversion Service

Currency Conversion Service (CCS) can convert a bucket of currencies into another currency. It uses the Forex Service to get current currency exchange values. CCS is the Service Consumer.

An example request and response is shown below:

GET to http://localhost:8100/currency-converter/from/EUR/to/INR/quantity/10000

{

id: 10002,

from: "EUR",

to: "INR",

conversionMultiple: 75,

quantity: 10000,

totalCalculatedAmount: 750000,

port: 8000,

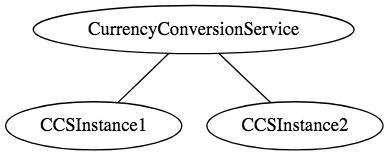
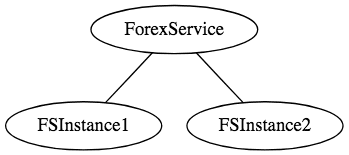
}

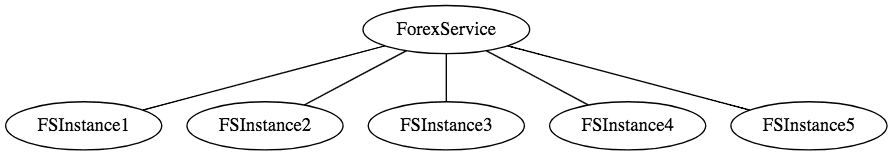
The request above is to find the value of 10000 EUR in INR. The totalCalculatedAmount is 750000 INR.

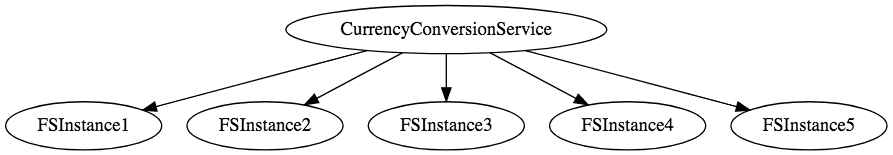
The diagram below shows the communication between CCS and FS.

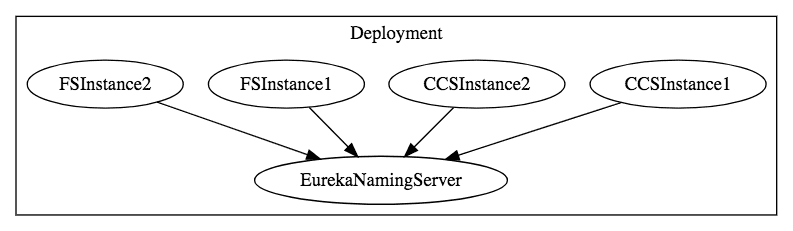


### Eureka Naming Server and Ribbon

Based on the load, we can have multiple instances of the Currency Conversion Service and the Forex Service running.  

And the number of instances for each service might vary with time. Below picture shows a specific instance where there are 5 instances of the Forex Service. 

What needs to happen in the above situation is load should be uniformly distributed among these 5 instances. 

In this series of articles, we will use Ribbon for Load Balancing and Eureka Naming server for registering all microservices. 

*Do not worry if you are not clear about a few things. The idea is to give a big picture before we get our hands dirty and create the microservices step by step*

### What is a Monolith Application?

Have you ever worked in a project

* Which is released (taken to production) once every few months
* Which has a wide range of features and functionality
* Which has a team of more than 50 working for it
* Where debugging problems is a big challenge
* Where bringing in new technology and new process is almost impossible

These are typical characteristics of a Monolith applications.

*Monolith applications are typically huge - more 100,000 line of code. In some instances even more than million lines of code.*

Monoliths are characterized by

* Large Application Size
* Long Release Cycles
* Large Teams

Typical Challenges include

* Scalability Challenges
* New Technology Adoption
* New Processes - Agile?
* Difficult to Automation Test
* Difficult ot Adapt to Modern Development Practices
* Adapting to Device Explosion

## Microservices

Microservice Architectures evolved as a solution to the scalability and innovotation challenges with Monolith architectures.

There are a number of definitions proposed for Microservices

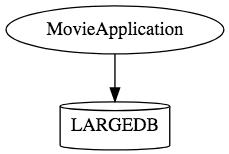
*Small autonomous services that work together - Sam Newman*

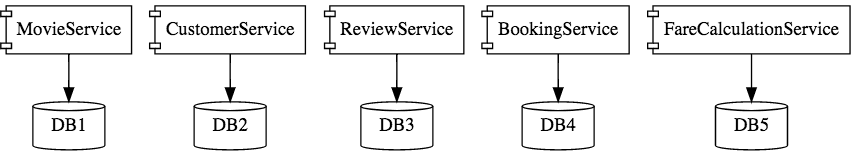
*Developing a single application as a suite of small services each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies - James Lewis and Martin Fowler*

While there is no single accepted definition for microservices, for me, there are a few important characteristics:

* REST - Built around RESTful Resources. Communication can be HTTP or event based.
* Small Well Chosen Deployable Units - Bounded Contexts
* Cloud Enabled - Dynamic Scaling

## How does Microservice Architecture look like?

This is how a monolith would look like. One application for everything. 

This is how the same application would look like when developed using Microservices Architecture. 

Microservice Architectures involve a number of small, well designed, components interacting with messages. https://www.springboottutorial.com/images/Microservices-Chain-Example.png

## Advantages of Microservices

Advantages

* New Technology & Process Adaption becomes easier. You can try new technologies with the newer microservices that we create.
* Faster Release Cycles
* Scaling with Cloud

## Challenges with Microservice Architectures

While developing a number of smaller components might look easy, there are a number of inherent complexities that are associated with microservices architectures.

Lets look at some of the challenges:

* Quick Setup needed : You cannot spend a month setting up each microservice. You should be able to create microservices quickly.
* Automation : Because there are a number of smaller components instead of a monolith, you need to automate everything - Builds, Deployment, Monitoring etc.
* Visibility : You now have a number of smaller components to deploy and maintain. Maybe 100 or maybe 1000 components. You should be able to monitor and identify problems automatically. You need great visibility around all the components.
* Bounded Context : Deciding the boundaries of a microservice is not an easy task. Bounded Contexts from Domain Driven Design is a good starting point. Your understanding of the domain evolves over a period of time. You need to ensure that the microservice boundaries evolve.
* Configuration Management : You need to maintain configurations for hundreds of components across environments. You would need a Configuration Management solution
* Dynamic Scale Up and Scale Down : The advantages of microservices will only be realized if your applications can scaled up and down easily in the cloud.
* Pack of Cards : If a microservice at the bottom of the call chain fails, it can have knock on effects on all other microservices. Microservices should be fault tolerant by Design.
* Debugging : When there is a problem that needs investigation, you might need to look into multiple services across different components. Centralized Logging and Dashboards are essential to make it easy to debug problems.
* Consistency : You cannot have a wide range of tools solving the same problem. While it is important to foster innovation, it is also important to have some decentralized governance around the languages, platforms, technology and tools used for implementing/deploying/monitoring microservices.

## Solutions to Challenges with Microservice Architectures

### Spring Boot

*Enable building production ready applications quickly*

Provide non-functional features

* embedded servers (easy deployment with containers)
* metrics (monitoring)
* health checks (monitoring)
* externalized configuration

### Spring Cloud

*Spring Cloud provides solutions to cloud enable your microservices. It leverages and builds on top of some of the Cloud solutions opensourced by Netflix (Netflix OSS).*

#### Important Spring Cloud Modules

Dynamic Scale Up and Down. Using a combination of

* Naming Server (Eureka)
* Ribbon (Client Side Load Balancing)
* Feign (Easier REST Clients)

Visibility and Monitoring with

* Zipkin Distributed Tracing
* Netflix API Gateway

Configuration Management with

* Spring Cloud Config Server

Fault Tolerance with

* Hystrix

## Microservice Series of Articles

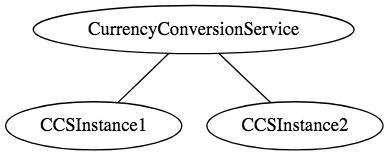
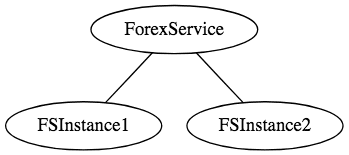
In this series of articles, we would create two microservices:

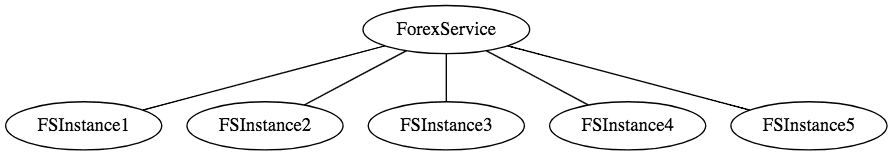
* Forex Service - Abbreviated as FS
* Currency Conversion Service - Abbreviated as CCS

The diagram below shows the communication between CCS and FS. We would establish communication between these two components.



We would want to be able to dynamically scale up and scale down the number of instances of each of these services.

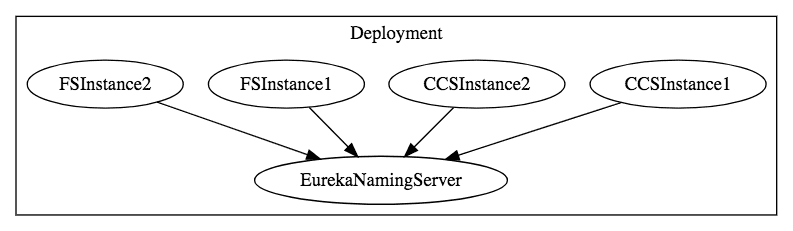
 

And the number of instances for each service might vary with time. Below picture shows a specific instance where there are 5 instances of the Forex Service. 

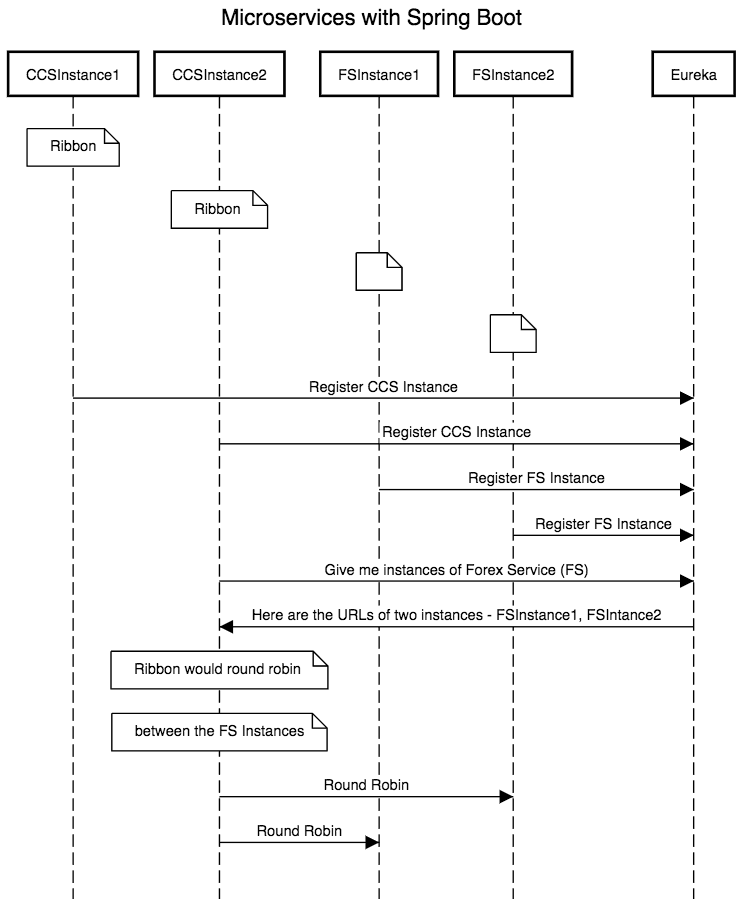
Implementing a solution for dynamic scale up and down needs to answer two questions

* How does the Currency Conversion Service (CCS) know how many instances of Forex Service (FS) are active?
* How does the Currency Conversion Service (CCS) distribute the load between the active instances.

Because we want this to be dynamic, we cannot hardcode the urls of FS in CCS. Thats why we bring in a Naming Server.



All instances of the components (CCS and FS) register with the Eureka Naming Server. When FS needs to call the CCS, it will ask Eureka Naming Server for the active instances. We will use Ribbon to do Client Side Load Balancing between the different instances of FS.

A high level sequence diagram of what would happen when there is a request from CCS to FS is shown below: 

Here are the next series of Articles

* Creating a Forex Microservice - We will create a simple rest service based on Spring Boot Starter Web and Spring Boot Started JPA. We will use Hibernate as JPA implmentation and connect to H2 database.
* Create the CCS - Currency Conversion Service - We will create a simple rest service using feign to invoke the Forex Microservice
* Use Ribbon for Load Balancing
* Implement Eureka Naming Service and connect FS and CCS through Eureka.