**Difference between an array and an ArrayList?**

First and Major difference between Array and ArrayList in Java is that Array is a fixed length data structure while ArrayList is a variable length Collection class. ... 4) One more major difference between ArrayList and Array is that, you can not store primitives in ArrayList, it can only contain Objects. While Array can contain both primitives and Objects in Java**.**

**Difference between an ArrayList and LinkedList?**

**List is ordered**

ArrayList **uses index based Array** LinkedList **is implemented using Doubly** Linked List **Data Structure.**

**HashTable is Synchronized and null value not allowed. HashMap is not Synchronized, allow one null value**

**Vector is Syncronized while ArrayList is not Syncronized.**

**Set and Map does not allow duplicates**

**Difference Between Daemon and User Threads**

Java offers two types of threads: user threads and daemon threads.

User threads are high-priority threads. The JVM will wait for any user thread to complete its task before terminating it.

On the other hand, daemon threads are low-priority threads whose only role is to provide services to user threads.

Since daemon threads are meant to serve user threads and are only needed while user threads are running, they won’t prevent the JVM from exiting once all user threads have finished their execution.

That’s why infinite loops, which typically exist in daemon threads, will not cause problems, because any code, including the finally blocks, won’t be executed once all user threads have finished their execution. For this reason, daemon threads are not recommended for I/O tasks.

**@SpringBootApplication**

Combination of three annotations: **@Configuration** which is used in Java-based configuration on Spring framework, **@ComponentScan** to enable component scanning of components you write e.g. [**@Controller**](http://javarevisited.blogspot.sg/2017/11/difference-between-component-service.html)classes, and **@EnableAutoConfgiuration**   
  
**Spring Boot autoconfiguration.** Some basic configurations are automatically configuredbased upon the JARs added in the classpath.

Suppose HSQLDB is present on your classpath, Spring will auto-configure an in-memory database for you. That mean those jars are downloaded using pom.xml, they are on class path.

Let’s say *spring-boot-web-starter* package then it can automatically configure dispatcher servlet and view resolver for you  
  
By default, this auto-configuration feature is not enabled and configure it by adding the @**EnableAutoConfiguration** or @**SpringBootApplicaiton** annotations to one of your @**Configuration** classes, generally the Main class which is used to run your application.

Binding the data and method into an class is encapsulation while hiding the internal implementation details of employee is abstraction.

Employee data and method are in employee class but calling employee methods in bank and display them is abstraction. Bank does not know how employee methods are implemented.

How Hash Map works

Implementation

To start using @Transactional annotation in a Spring based application, we need to first enable annotations in our Spring application by adding the needed configuration into spring context file –

|  |  |
| --- | --- |
| 1 | <tx:annotation-driven transaction-manager="txManager"/> |

Next is to define the transaction manager bean, with the same name as specified in the above transaction-manager attribute value.

The transaction managers could be –

2.1 DataSource Transaction manager

|  |  |
| --- | --- |
| 1  2  3  4 | <bean id="txManager"          class="org.springframework.jdbc.datasource.DataSourceTransactionManager">      <property name="dataSource" ref= "datasource" />  </bean> |

2.2 Hibernate Transaction manager

|  |  |
| --- | --- |
| 1  2  3  4 | <bean id="txManager"          class="org.springframework.orm.hibernate3.HibernateTransactionManager">      <property name="sessionFactory" ref= "sessionFactory" />  </bean> |

2.3 JPA Transaction manager

|  |  |
| --- | --- |
| 1  2  3  4 | <bean id="txManager"          class="org.springframework.orm.jpa.JpaTransactionManager">      <property name="entityManagerFactory" ref="entityManagerFactory" />  </bean> |

We are now ready to use @Transactional annotation either at the class or method level.

|  |  |
| --- | --- |
| 1  2  3  4 | @Transactional(value = "myTransactionManager", propagation = Propagation.REQUIRED, readOnly = true)  public void myMethod() {      ...  } |

**Collections in Java**

A Collection is a group of individual objects represented as a single unit. Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.

The Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are the two main “root” interfaces of Java collection classes.

**Need for Collection Framework :**   
Before Collection Framework (or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were Arrays or Vectors or Hashtables. All of these collections had no common interface.

// Java program to show whey collection framework was needed

import java.io.\*;

import java.util.\*;

class Test

{

public static void main (String[] args)

{

// Creating instances of array, vector and hashtable

int arr[] = new int[] {1, 2, 3, 4};

Vector<Integer> v = new Vector();

Hashtable<Integer, String> h = new Hashtable();

v.addElement(1);

v.addElement(2);

h.put(1,"geeks");

h.put(2,"4geeks");

// Array instance creation requires [], while Vector

// and hastable require ()

// Vector element insertion requires addElement(), but

// hashtable element insertion requires put()

// Accessing first element of array, vector and hashtable

System.out.println(arr[0]);

System.out.println(v.elementAt(0));

System.out.println(h.get(1));

// Array elements are accessed using [], vector elements

// using elementAt() and hashtable elements using get()

}

}

Accessing elements of these Data Structures was a hassle as each had a different method (and syntax) for accessing its members:  
close

|  |
| --- |
|  |

***chevron\_right***

*filter\_none*

Output:

1

1

geek

As we can see, none of these collections (Array, Vector or Hashtable) implement a standard member access interface. It was very difficult for programmers to write algorithms that can work for all kinds of Collections. Another drawback being that most of the ‘Vector’ methods are final, meaning we cannot extend the ’Vector’ class to implement a similar kind of Collection.  
***Java developers decided to come up with a common interface to deal with the above mentioned problems and introduced the Collection Framework in JDK 1.2***.

Both legacy Vectors and Hashtables were modified to conform to the Collection Framework.

**Advantages of Collection Framework:**

1. Consistent API : The API has a basic set of interfaces like Collection, Set, List, or Map. All classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have *some* common set of methods.
2. Reduces programming effort: A programmer doesn’t have to worry about the design of Collection, and he can focus on its best use in his program.
3. Increases program speed and quality: Increases performance by providing high-performance implementations of useful data structures and algorithms.

**Hierarchy of Collection Framework**

Collection Map

/ / \ \ |

/ / \ \ |

Set List Queue Dequeue SortedMap

/

/

SortedSet

**Core Interfaces in Collections**

Note that this diagram only shows core interfaces.

**Collection :** Root interface with basic methods like add(), remove(),

contains(), isEmpty(), addAll(), ... etc.

[**Set**](https://www.geeksforgeeks.org/set-in-java/) **:** Doesn't allow duplicates. Example implementations of Set

interface are HashSet (Hashing based) and TreeSet (balanced

BST based). Note that TreeSet implements **SortedSet**.

[**List**](https://www.geeksforgeeks.org/list-interface-java-examples/) **:** Can contain duplicates and elements are ordered. Example

implementations are LinkedList (linked list based) and

[ArrayList](https://www.geeksforgeeks.org/array-vs-arraylist-in-java/) (dynamic array based)

[**Queue**](https://www.geeksforgeeks.org/queue-interface-java/) **:** Typically order elements in FIFO order except exceptions

like PriorityQueue.

[**Deque**](https://www.geeksforgeeks.org/deque-interface-java-example/) **:** Elements can be inserted and removed at both ends. Allows

both LIFO and FIFO.

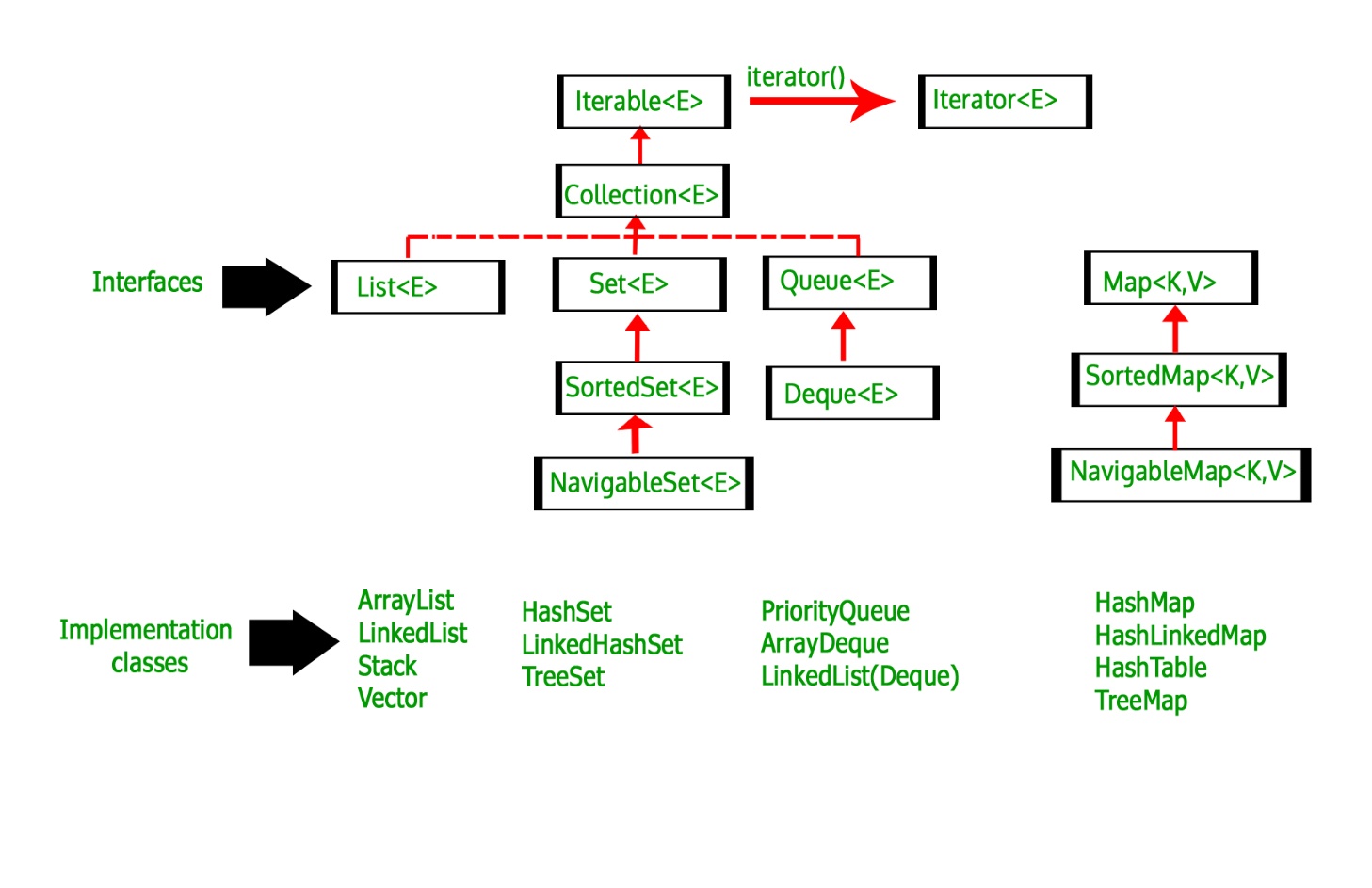
[**Map**](https://www.geeksforgeeks.org/map-interface-java-examples/) **:** Contains Key value pairs. Doesn't allow duplicates. Example

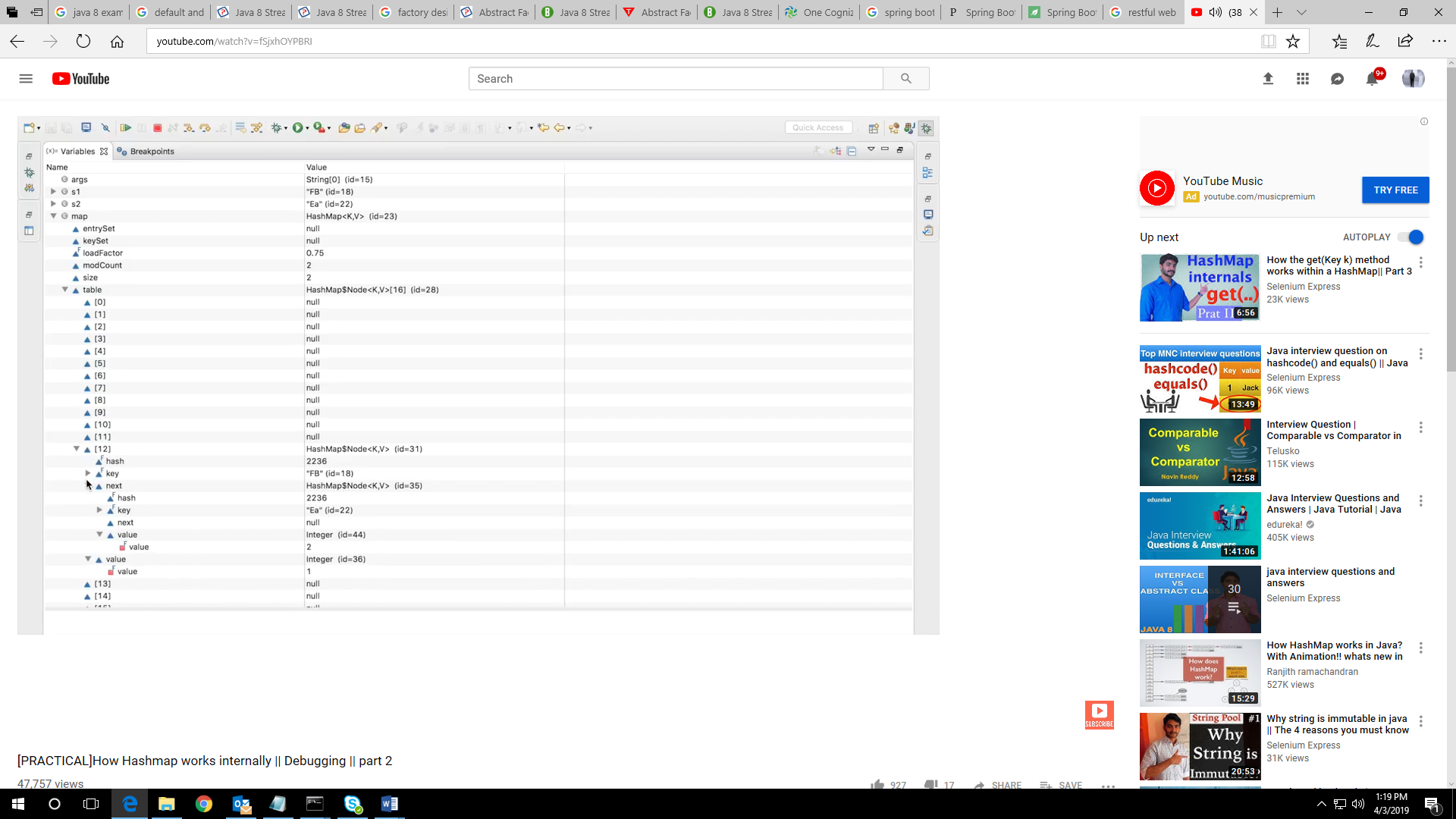
implementation are [HashMap](http://www.geeksforgeeks.org/java-util-hashmap-in-java/) and TreeMap.

[TreeMap](https://www.geeksforgeeks.org/treemap-in-java/) implements **SortedMap**.

The difference between Set and Map interface is that in Set we

have only keys, whereas in Map, we have key, value pairs.





## ****How hashCode and equals method works****

**package** com.java8practice.collection;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** HashCodeEquals {

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

Employee emp1=**new** Employee(1);

Employee emp2=**new** Employee(1);

Map<Employee,String> map1=**new** HashMap<Employee,String>();

map1.put(emp1, "one");

map1.put(emp2, "one");

//Output 2 as //hashCode and equals method is not overriden in employee class which checks equals method of Object class where references are are checked which is 2

//If we override hashCode and equals method of employee class then output becomes 1 as both the objects hasve same value "one".

System.***out***.println(map1.size());

Integer i1=**new** Integer(1); //Output 1 as hashCode and equals method is overriden in wrapper classes which checks value

Integer i2=**new** Integer(1);

Map<Integer,String> map2=**new** HashMap<Integer,String>();

map2.put(i1, "one");

map2.put(i2, "one");

System.***out***.println(map2.size());

}

## }

## Employee.java

**package** com.java8practice.collection;

**public** **class** Employee {

**int** id;

/\*public Employee(int id){

this.id=id;

}\*/

**public** Employee(**int** id2) {

**this**.id=id2;

}

@Override

**public** **int** hashCode() {

**final** **int** prime = 31;

**int** result = 1;

result = prime \* result + id;

**return** result;

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

Employee other = (Employee) obj;

**if** (id != other.id)

**return** **false**;

**return** **true**;

}

}

**CallableFuture**

Callable is a thread that returns the result. There is a method call() in Callable interface that must be overridden for computation task. To run Callable, submit() method of ExecutorService is used. ExecutorService also provides invokeAll() and invokeAny () method to run Callable threads. To fetch the result of call() method of Callable interface, java provides Future class. ExecutorService.submit() method returns [Future](https://www.concretepage.com/java/java-future-example) instance and then get() method of Future, returns the result of call() method of Callable.

## call() in Callable Interface

call() method returns the result. The result can be object of the class which has been declared in class declaration. The basic purpose of call() method is computation of result. This is same as run() method of Runnable interface but returns the result.

## How to Run Callable in Java

To use Callable, we are creating a class which is implementing Callable interface. The class is overriding call() method.   
**CallableDemo.java**

package com.concretepage.util.concurrent;

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutionException;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

public class CallableDemo {

public static void main(String[] args) throws InterruptedException, ExecutionException {

ExecutorService service = Executors.newSingleThreadExecutor();

SumTask sumTask = new SumTask(20);

Future<Integer> future = service.submit(sumTask);

Integer result = future.get();

System.out.println(result);

}

}

class SumTask implements Callable<Integer> {

private int num = 0;

public SumTask(int num){

this.num = num;

}

@Override

public Integer call() throws Exception {

int result = 0;

for(int i=1;i<=num;i++){

result+=i;

}

return result;

}

}

## ****Overview****

In this quick tutorial, you’ll learn about the different types of bean scopes in the Spring framework.

The scope of a bean defines the life cycle and visibility of that bean in the contexts in which it is used.

The latest version of Spring framework defines 6 types of scopes:

* singleton
* prototype
* request
* session
* application
* websocket

The last four scopes mentioned request, session, application and websocket are only available in a web-aware application.

## ****2. Singleton Scope****

Defining a bean with singleton scope means the container creates a single instance of that bean, and all requests for that bean name will return the same object, which is cached. Any modifications to the object will be reflected in all references to the bean. This scope is the default value if no other scope is specified.

Let’s create a Person entity to exemplify the concept of scopes:

[?](https://www.baeldung.com/spring-bean-scopes)

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class Person {    private String name;      // standard constructor, getters and setters  } |

Afterwards, we define the bean with singleton scope by using the @Scope annotation:

[?](https://www.baeldung.com/spring-bean-scopes)

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Bean  @Scope("singleton")  public Person personSingleton() {      return new Person();  } |

We can also use a constant instead of the String value in the following manner:

[?](https://www.baeldung.com/spring-bean-scopes)

|  |  |
| --- | --- |
| 1 | @Scope(value = ConfigurableBeanFactory.SCOPE\_SINGLETON) |

Now we proceed to write a test that shows that two objects referring to the same bean will have the same values, even if only one of them changes their state, as they are both referencing the same bean instance:

[?](https://www.baeldung.com/spring-bean-scopes)

|  |  |
| --- | --- |
|  | private static final String NAME = "John Smith";    @Test  public void givenSingletonScope\_whenSetName\_thenEqualNames() {      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("scopes.xml");        Person personSingletonA = (Person) applicationContext.getBean("personSingleton");      Person personSingletonB = (Person) applicationContext.getBean("personSingleton");        personSingletonA.setName(NAME);      Assert.assertEquals(NAME, personSingletonB.getName());        ((AbstractApplicationContext) applicationContext).close();  } |

The scopes.xml file in this example should contain the xml definitions of the beans used:

[?](https://www.baeldung.com/spring-bean-scopes)

|  |  |
| --- | --- |
| 1  2 | <?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="personSingleton" class="org.baeldung.scopes.Person" scope="singleton"/>  </beans> |

## ****3. Prototype Scope****

A bean with prototype scope will return a different instance every time it is requested from the container. It is defined by setting the value prototype to the *@Scope* annotation in the bean definition:

|  |  |
| --- | --- |
|  | @Bean  @Scope("prototype")  public Person personPrototype() {      return new Person();  } |

We could also use a constant as we did for the singleton scope:

|  |  |
| --- | --- |
|  | @Scope(value = ConfigurableBeanFactory.SCOPE\_PROTOTYPE) |

We will now write a similar test as before that shows two objects requesting the same bean name with scope prototype will have different states, as they are no longer referring to the same bean instance:

|  |  |
| --- | --- |
|  | private static final String NAME = "John Smith";  private static final String NAME\_OTHER = "Anna Jones";  @Test  public void givenPrototypeScope\_whenSetNames\_thenDifferentNames() {      ApplicationContext applicationContext = new ClassPathXmlApplicationContext("scopes.xml");        Person personPrototypeA = (Person) applicationContext.getBean("personPrototype");      Person personPrototypeB = (Person) applicationContext.getBean("personPrototype");        personPrototypeA.setName(NAME);      personPrototypeB.setName(NAME\_OTHER);        Assert.assertEquals(NAME, personPrototypeA.getName());      Assert.assertEquals(NAME\_OTHER, personPrototypeB.getName());        ((AbstractApplicationContext) applicationContext).close();  } |

The scopes.xml file is similar to the one presented in the previous section while adding the xml definition for the bean with prototype scope:

|  |  |
| --- | --- |
|  | <bean id="prsonPrototype" class="org.baeldung.scopes.Person" scope="prototype"/> |

## ****4. Web Aware Scopes****

As mentioned, there are four additional scopes that are only available in a web-aware application context. These are less often used in practice.

The request scope creates a bean instance for a single HTTP request while session scope creates for an HTTP Session.

The application scope creates the bean instance for the lifecycle a ServletContext and the websocket scope creates it for a particular WebSocket session.

Let’s create a class to use for instantiating the beans:

|  |  |
| --- | --- |
|  | public class HelloMessageGenerator {      private String message;        // standard getter and setter  } |

### ****4.1. Request Scope****

We can define the bean with request scope using the @Scope annotation:

|  |  |
| --- | --- |
|  | @Bean  @Scope(value = WebApplicationContext.SCOPE\_REQUEST, proxyMode = ScopedProxyMode.TARGET\_CLASS)  public HelloMessageGenerator requestScopedBean() {      return new HelloMessageGenerator();  } |

The proxyMode attribute is necessary because, at the moment of the instantiation of the web application context, there is no active request. Spring will create a proxy to be injected as a dependency, and instantiate the target bean when it is needed in a request.

Next, we can define a controller that has an injected reference to the requestScopedBean. We need to access the same request twice in order to test the web specific scopes.

If we display the message each time the request is run, we can see that the value is reset to null, even though it is later changed in the method. This is because of a different bean instance being returned for each request.

[?](https://www.baeldung.com/spring-bean-scopes)

|  |  |
| --- | --- |
| 1 | @Controller  public class ScopesController {      @Resource(name = "requestScopedBean")      HelloMessageGenerator requestScopedBean;        @RequestMapping("/scopes/request")      public String getRequestScopeMessage(final Model model) {          model.addAttribute("previousMessage", requestScopedBean.getMessage());          requestScopedBean.setMessage("Good morning!");          model.addAttribute("currentMessage", requestScopedBean.getMessage());          return "scopesExample";      }  } |

### ****4.2. Session Scope****

We can define the bean with *session* scope in a similar manner:

|  |  |
| --- | --- |
|  | @Bean  @Scope(value = WebApplicationContext.SCOPE\_SESSION, proxyMode = ScopedProxyMode.TARGET\_CLASS)  public HelloMessageGenerator sessionScopedBean() {      return new HelloMessageGenerator();  } |

Next, we define a controller with a reference to the sessionScopedBean. Again, we need to run two requests in order to show that the value of the message field is the same for the session.

In this case, when the request is made for the first time, the value message is null. But once, it is changed, then that value is retained for subsequent requests as the same instance of the bean is returned for the entire session.

|  |  |
| --- | --- |
|  | @Controller  public class ScopesController {      @Resource(name = "sessionScopedBean")      HelloMessageGenerator sessionScopedBean;        @RequestMapping("/scopes/session")      public String getSessionScopeMessage(final Model model) {          model.addAttribute("previousMessage", sessionScopedBean.getMessage());          sessionScopedBean.setMessage("Good afternoon!");          model.addAttribute("currentMessage", sessionScopedBean.getMessage());          return "scopesExample";      }  } |

### ****4.3. Application Scope****

The application scope creates the bean instance for the lifecycle of a ServletContext.

This is similar to the singleton scope but there is a very important difference with regards to the scope of the bean.

When beans are application scoped the same instance of the bean is shared across multiple servlet-based applications running in the same ServletContext, while singleton-scoped beans are scoped to a single application context only.

Let’s create the bean with application scope:

|  |  |
| --- | --- |
|  | @Bean  @Scope(value = WebApplicationContext.SCOPE\_APPLICATION, proxyMode = ScopedProxyMode.TARGET\_CLASS)  public HelloMessageGenerator applicationScopedBean() {      return new HelloMessageGenerator();  } |

And the controller that references this bean:

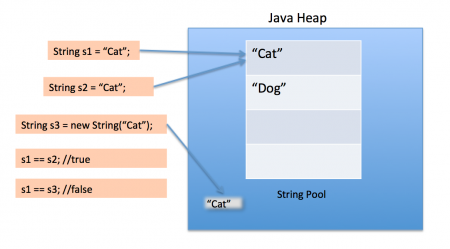
|  |  |
| --- | --- |
|  | @Controller  public class ScopesController {      @Resource(name = "applicationScopedBean")      HelloMessageGenerator applicationScopedBean;       @RequestMapping("/scopes/application")      public String getApplicationScopeMessage(final Model model) {          model.addAttribute("previousMessage", applicationScopedBean.getMessage());          applicationScopedBean.setMessage("Good afternoon!");          model.addAttribute("currentMessage", applicationScopedBean.getMessage());          return "scopesExample";      }  } |

**What is Java String Pool?**

**String Pool in java** is a pool of Strings stored in [**Java Heap Memory**](https://www.journaldev.com/4098/java-heap-space-vs-stack-memory). We know that String is special class in java and we can create String object using new operator as well as providing values in double quotes.

## String Pool in Java

Here is a diagram which clearly explains how String Pool is maintained in [java heap](https://www.journaldev.com/4098/java-heap-space-vs-stack-memory) space and what happens when we use different ways to create Strings.

[](https://cdn.journaldev.com/wp-content/uploads/2012/11/String-Pool-Java1.png)

String Pool is possible only because [String is immutable in Java](https://www.journaldev.com/802/string-immutable-final-java) and its implementation of [String interning](https://en.wikipedia.org/wiki/String_interning) concept. String pool is also example of [Flyweight design pattern](https://www.journaldev.com/1562/flyweight-design-pattern-java).

String pool helps in saving a lot of space for Java Runtime although it takes more time to create the String.

When we use double quotes to create a String, it first looks for String with the same value in the String pool, if found it just returns the reference else it creates a new String in the pool and then returns the reference.

However using new operator, we force String class to create a new String object in heap space. We can use intern() method to put it into the pool or refer to another String object from the string pool having the same value.

package com.journaldev.util;

public class StringPool {

/\*\*

\* [Java String](https://www.journaldev.com/16928/java-string) Pool example

\* @param args

\*/

public static void main(String[] args) {

String s1 = "Cat";

String s2 = "Cat";

String s3 = new String("Cat");

System.out.println("s1 == s2 :"+(s1==s2));

System.out.println("s1 == s3 :"+(s1==s3));

}

}

Output of the above program is:

Copy

s1 == s2 :true

s1 == s3 :false

## How many Strings are getting Created in the String Pool?

Sometimes in [java interview](https://www.journaldev.com/2366/core-java-interview-questions-and-answers), you will be asked a question around String pool. For example, how many strings are getting created in the below statement;**Copy**

String str = new String("Cat");

In the above statement, either 1 or 2 string will be created. If there is already a string literal “Cat” in the pool, then only one string “str” will be created in the pool. If there is no string literal “Cat” in the pool, then it will be first created in the pool and then in the heap space, so a total of 2 string objects will be created.

Generics are used to create Generic Classes and Generic methods which can work with different Types(Classes).

### Why do we need Generics? Can you give an example of how Generics make a program more flexible?

Consider the class below:

class MyList {

private List<String> values;

void add(String value) {

values.add(value);

}

void remove(String value) {

values.remove(value);

}

}

MyList can be used to store a list of Strings only.

MyList myList = new MyList();

myList.add("Value 1");

myList.add("Value 2");

To store integers, we need to create a new class. This is problem that Generics solve. Instead of hard-coding String class as the only type the class can work with, we make the class type a parameter to the class.

##### Example with Generics

Let’s replace String with T and create a new class. Now the MyListGeneric class can be used to create a list of Integers or a list of Strings

class MyListGeneric<T> {

private List<T> values;

void add(T value) {

values.add(value);

}

void remove(T value) {

values.remove(value);

}

T get(int index) {

return values.get(index);

}

}

MyListGeneric<String> myListString = new MyListGeneric<String>();

myListString.add("Value 1");

myListString.add("Value 2");

MyListGeneric<Integer> myListInteger = new MyListGeneric<Integer>();

myListInteger.add(1);

myListInteger.add(2);

### How do you declare a Generic Class?

Note the declaration of class:Instead of T, We can use any valid identifier.

class MyListGeneric<T>

### What are the restrictions in using generic type that is declared in a class declaration?

If a generic is declared as part of class declaration, it can be used any where a type can be used in a class - method (return type or argument), member variable etc. For Example: See how T is used as a parameter and return type in the class MyListGeneric.

### How can we restrict Generics to a subclass of particular class?

In MyListGeneric, Type T is defined as part of class declaration. Any Java Type can be used a type for this class. If we would want to restrict the types allowed for a Generic Type, we can use a Generic Restrictions. Consider the example class below: In declaration of the class, we specified a constraint "T extends Number". We can use the class MyListRestricted with any class extending (any sub class of) Number - Float, Integer, Double etc.

class MyListRestricted<T extends Number> {

private List<T> values;

void add(T value) {

values.add(value);

}

void remove(T value) {

values.remove(value);

}

T get(int index) {

return values.get(index);

}

}

MyListRestricted<Integer> restrictedListInteger = new MyListRestricted<Integer>();

restrictedListInteger.add(1);

restrictedListInteger.add(2);

String not valid substitute for constraint "T extends Number".

//MyListRestricted<String> restrictedStringList =

// new MyListRestricted<String>();//COMPILER ERROR