**Lead tester**

* Define and execute the automation strategy
* Proficient in analyses of automation framework i.e. functional & microservice automation
* Develop new frameworks best suited to the requirement
* Take a lead in fault fixing and help other members to move forward.
* Clarity on Java concepts and hands on experience to implement the same
* Suggest and execute new ideas of left shifting the testing in AGILE methodology.
* Suggest new methodology\framework like BDD and substantiate benefits for stakeholders.

**Automation engg**

* Proficient in automation script development
* Independently maintain the automation suite
* A good understanding of Java concepts
* Assist lead tester to find out the shortfalls of the framework at proposal stage
* Able to implement new libraries in the framework.
* Hands on experience on BDD and trending frameworks.

1. String is immutable - why ?

2. Collections, About Hash Map and its benefits with real time project example

3. Method overloading in Selenium

3. Scenario based question - For a customer the element looks like drop down, but when inspected it doesn't have any attribute and also doesn't have

any parent element or labels, How to automate and click on a particular element.

4. How much % of Regression cases will be automated.

5. BDD framework

6. TestNg - Name few annotations

7. TestNg - DataProvider

8. how will you handle the situation if an technical issues is found and there is no technical person to help

9. Cross Browser testing - Framework to follow and what are the challenges

10. Mobile testing

11. Bitbucket - Basic commands which we use daily

12. Bitbukcet - How to handle the clashes when 3 persons push the code to repository at 3 different timings

13. Github basics

14. CI/CD basics

15. How will you handle the defect which occurs in CI/CD

16. Tools used in CI/CD

17. How will you send Data to your Framework

18. Read data from excel – Explain the class and objects used

19. SQL connection – How to achieve it in selenium and what are the issues and challenges faced

**Lead tester**

* Define and execute the automation strategy
* Proficient in analyses of automation framework i.e. functional & microservice automation
* Develop new frameworks best suited to the requirement
* Take a lead in fault fixing and help other members to move forward.
* Clarity on Java concepts and hands on experience to implement the same
* Suggest and execute new ideas of left shifting the testing in AGILE methodology.
* **Shift Left** is a practice intended to find and prevent defects early in the software delivery process. The idea is to improve quality by moving tasks to the **left** as early in the lifecycle as possible. **Shift Left testing** means **testing** earlier in the software development process.
* Suggest new methodology\framework like BDD and substantiate benefits for stakeholders.

**Automation engg**

* Proficient in automation script development
* Independently maintain the automation suite
* A good understanding of Java concepts
* Assist lead tester to find out the shortfalls of the framework at proposal stage
* Able to implement new libraries in the framework.
* Hands on experience on BDD and trending frameworks.

1. String is immutable - why ?

Uses a concept of String pool, so when the string is assigned with other variable, the value of string doest changed, only the reference points to new value and old value is still in string pool.

2. Collections, About Hash Map and its benefits with real time project example

3. Method overloading in Selenium

3. Scenario based question - For a customer the element looks like drop down, but when inspected it doesn't have any attribute and also doesn't have

any parent element or labels, How to automate and click on a particular element.

4. How much % of Regression cases will be automated.

5. BDD framework

6. TestNg - Name few annotations

@Test

@Parameters, @Listeners, @Factory, @Dataprovider, @BeforeSuite,@BeforeClass,@BeforeTest,@BeforeMEthod,@BeforeGroup

@Test(priority = 3)

@Parameters({ "suite-param" })

@Test

7. TestNg – DataProvider

TestNG functionality to import data from Excel or other sources.

We need to create a method for dataprovider with a name.

We will call using the @Dataprovider = Dataprovidername and we

8. how will you handle the situation if an technical issues is found and there is no technical person to help

Never have a dependency

9. Cross Browser testing - Framework to follow and what are the challenges

TestNG is the best framework for Crossbrowser testing, we also have parallel execution for different browsers where we have to update the Suite with parallel as Tests\Methods.

10. Mobile testing

11. Bitbucket - Basic commands which we use daily

12. Bitbukcet - How to handle the clashes when 3 persons push the code to repository at 3 different timings

13. Github basics

Version control. Have used the GITHUB Window for automatically managing the codes.

14. CI/CD basics

15. How will you handle the defect which occurs in CI/CD

Using the Console log in Jenkins, issues in build or run can be verified.

16. Tools used in CI/CD

Jenkins

17. How will you send Data to your Framework

18. Read data from excel – Explain the class and objects used

19. SQL connection – How to achieve it in selenium and what are the issues and challenges faced

**Quality standards** are **defined** as documents that provide **requirements**, specifications, guidelines, or characteristics that can be used consistently to ensure that materials, products, processes, and services are fit for their purpose.

What is the importance of quality standards?

It's about safety, delivering on a promise and meeting the very basics of customer expectations. But, by meeting **quality standards**, companies often reap better profits and reduce losses. Those that exceed **quality standards** stand out above their competitors and further their potential for profit and consumer loyalty.

**KEY DIFFERENCE**

* Verification process includes checking of documents, design, code and program whereas Validation process includes testing and validation of the actual product.
* Verification does not involve code execution while Validation involves code execution.
* Verification uses methods like reviews, walkthroughs, inspections and desk-checking whereas Validation uses methods like black box testing, white box testing and non-functional testing.
* Verification checks whether the software confirms a specification whereas Validation checks whether the software meets the requirements and expectations.
* Verification finds the bugs early in the development cycle whereas Validation finds the bugs that verification can not catch.
* Verification process targets on software architecture, design, database, etc. while Validation process targets the actual software product.
* Verification is done by the QA team while Validation is done by the involvement of testing team with QA team.
* Verification process comes before validation whereas Validation process comes after verification.

Asserts

Assert.assertEquals(ExpectedTitle, ActualTitle);

Assert.assertNotEquals(ActualTitle, ExpectedTitle);

Boolean verifyTitle = driver.getTitle().equalsIgnoreCase("Most Reliable App & Cross Browser Testing Platform | BrowserStack");

assertTrue(verifyTitle);

Boolean verifyTitle = driver.getTitle().equalsIgnoreCase("Most Reliable App & Cross Browser Testing Platform");

assertFalse(verifyTitle);

driver.navigate().to("https://www.browserstack.com/");

String verifyAssertNull = null;

assertNull(verifyAssertNull);

Boolean verifyTitle = driver.getTitle().equalsIgnoreCase("Most Reliable App & Cross Browser Testing Platform");

assertNotNull(verifyTitle);

assertEquals(ExpectedTitle, ActualTitle);

assertNotEquals(ExpectedTitle, "browserstack");

assertTrue(verifyTitleIsPresent);

assertFalse(verifyTitleIsChanged);

assertNotNull(verifyTitleIsPresent);

assertNull(verifyAssertNull);

@Test

    @Parameters ({"val1", "val2"})

    public void mul(int v1, int v2) {

     int prod = v1\*v2;

        System.out.println("The Product Of Value 1 and 2 is " + prod);

    }

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">  <suite name="TestNG Parameters Suite">     <parameter name="val1" value="3" />     <parameter name="val2" value="50" />     <test name="Params">        <classes>           <class name="Params" />        </classes>     </test>     <test name="Multiply">        <classes>           <class name="Multiply" />        </classes>     </test>  </suite> |

@Test (dependsOnMethods = { "OpenBrowser" })

  public void SignIn() {

  System.out.println("This will execute second (SignIn)");

  }

  @Test

  public void OpenBrowser() {

  System.out.println("This will execute first (Open Browser)");

  }

}

@Test(dataProvider="SearchProvider")

@DataProvider(name="SearchProvider")

public Object[][] getDataFromDataprovider(){

return new Object[][]

{

{ "Guru99", "India" },

{ "Krishna", "UK" },

{ "Bhupesh", "USA" }

};

**Advanced JAVA**

# **Java Enumerations**

**Enumeration** means a list of named constant. In Java, enumeration defines a class type. An Enumeration can have constructors, methods and instance variables. It is created using **enum** keyword. Each enumeration constant is *public*, *static* and *final* by default. Even though enumeration defines a class type and have constructors, you do not instantiate an **enum** using **new**. Enumeration variables are used and declared in much a same way as you do a primitive variable.

**enum** WeekDays{

***SUNDAY***, ***MONDAY***, ***TUESDAY***, ***WEDNESDAY***, ***THURSDAY***, ***FRIDAY***, ***SATURDAY***

}

**class** Learning1

{

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

{

WeekDays wk; //wk is an enumeration variable of type WeekDays

wk = WeekDays.***SUNDAY***; //wk can be assigned only the constants defined under enum type Weekdays

System.***out***.println("Today is "+wk);

}

}

enum WeekDays{

SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY

}

class Demo {

public static void main(String args[])

{

WeekDays weekDays = WeekDays.WEDNESDAY;

if(weekDays == WeekDays.SUNDAY || weekDays == WeekDays.SATURDAY)

System.out.println("It is Weekend");

else

System.out.println("It is weekday: "+weekDays);

}

}

### **Traversing Enumeration Elements**

We can iterate enumeration elements by calling its static method values(). This method returns an array of all the enum constants that further can be iterate using for loop. See the below example.

enum WeekDays{

SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY

}

class Demo {

public static void main(String args[])

{

WeekDays[] weekDays = WeekDays.values();

for(WeekDays weekday : weekDays ){ System.out.println(weekday);

} } }

**CONSTRUCTOR**

Constructor is a block of code that initializes the newly created object. A constructor resembles an instance method in java but it’s not a method as it doesn’t have a return type. In short constructor and method are different(More on this at the end of this guide). People often refer constructor as special type of method in Java.

Constructor has same name as the class and looks like this in a java code.

public class MyClass{

//This is the constructor

MyClass(){

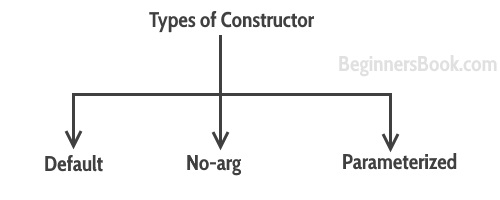
}

..

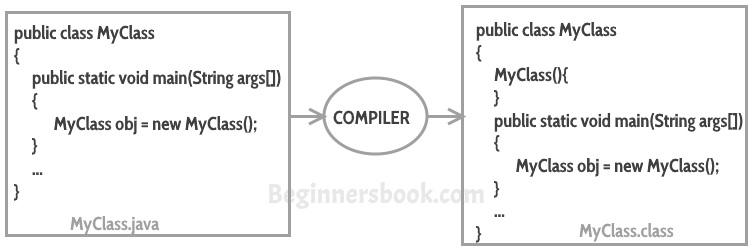
}

Note that the constructor name matches with the class name and it doesn’t have a return type.

## Types of Constructors

There are three types of constructors: Default, No-arg constructor and Parameterized.  


### Default constructor

If you do not implement any constructor in your class, Java compiler inserts a [default constructor](https://beginnersbook.com/2014/01/default-constructor-java-example/) into your code on your behalf. This constructor is known as default constructor. You would not find it in your source code(the java file) as it would be inserted into the code during compilation and exists in .class file. This process is shown in the diagram below:  


If you implement any constructor then you no longer receive a default constructor from Java compiler.

### no-arg constructor:

Constructor with no arguments is known as **no-arg constructor**. The signature is same as default constructor, however body can have any code unlike default constructor where the body of the constructor is empty.

Although you may see some people claim that that default and no-arg constructor is same but in fact they are not, even if you write **public Demo() { }** in your class Demo it cannot be called default constructor since you have written the code of it.

#### Example: no-arg constructor

class Demo

{

public Demo()

{

System.out.println("This is a no argument constructor");

}

public static void main(String args[]) {

new Demo();

}

}

Output:  
This is a no argument constructor

### Parameterized constructor

Constructor with arguments(or you can say parameters) is known as [Parameterized constructor](https://beginnersbook.com/2014/01/parameterized-constructor-in-java-example/).

#### Example: parameterized constructor

In this example we have a parameterized constructor with two parameters id and name. While creating the objects obj1 and obj2 I have passed two arguments so that this constructor gets invoked after creation of obj1 and obj2.

public class Employee {

int empId;

String empName;

//parameterized constructor with two parameters

Employee(int id, String name){

this.empId = id;

this.empName = name;

}

void info(){

System.out.println("Id: "+empId+" Name: "+empName);

}

public static void main(String args[]){

Employee obj1 = new Employee(10245,"Chaitanya");

Employee obj2 = new Employee(92232,"Negan");

obj1.info();

obj2.info();

}

}

**Output:**

Id: 10245 Name: Chaitanya

Id: 92232 Name: Negan

The this keyword refers to the current object in a method or constructor.

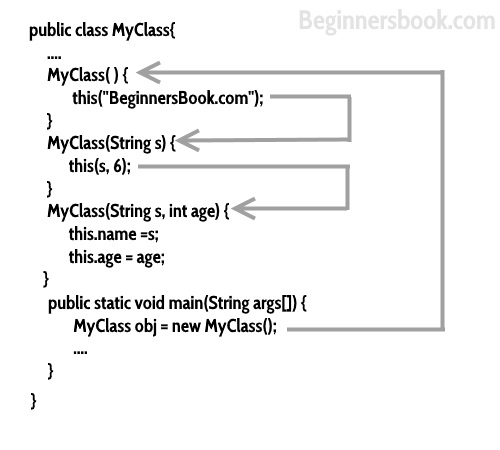
The most common use of the this keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter). If you omit the keyword in the example above, the output would be "0" instead of "5".

this can also be used to:

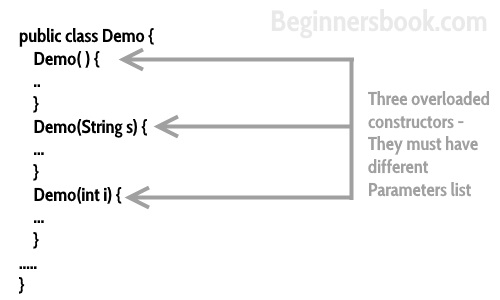
* Invoke current class constructor
* Invoke current class method
* Return the current class object
* Pass an argument in the method call
* Pass an argument in the constructor call

## super keyword is used to access methods of the parent class while this is used to access methods of the current class. this keyword. this is a reserved keyword in java i.e, we can't use it as an identifier. this is used to refer current-class's instance as well as static members.

## Constructor Chaining

When A constructor calls another constructor of same class then this is called constructor chaining. Read more about it [here](https://beginnersbook.com/2013/12/java-constructor-chaining-with-example/).  


## Constructor Overloading

Constructor overloading is a concept of having more than one constructor with different parameters list, in such a way so that each constructor performs a different task.  


Refer [constructor overloading with example](https://beginnersbook.com/2013/05/constructor-overloading/) for more details with example.

## Java Copy Constructor

A copy constructor is used for copying the values of one object to another object.

class JavaExample{

String web;

JavaExample(String w){

web = w;

}

/\* This is the Copy Constructor, it

\* copies the values of one object

\* to the another object (the object

\* that invokes this constructor)

\*/

JavaExample(JavaExample je){

web = je.web;

}

void disp(){

System.out.println("Website: "+web);

}

public static void main(String args[]){

JavaExample obj1 = new JavaExample("BeginnersBook");

/\* Passing the object as an argument to the constructor

\* This will invoke the copy constructor

\*/

JavaExample obj2 = new JavaExample(obj1);

obj1.disp();

obj2.disp();

}

}

Output:

Website: BeginnersBook

Website: BeginnersBook

## Quick Recap

1. Every class has a constructor whether it’s a normal class or a abstract class.
2. Constructors are not methods and they don’t have any return type.
3. Constructor name should match with class name .
4. Constructor can use any access specifier, they can be declared as private also. Private constructors are possible in java but there scope is within the class only.
5. **Like constructors method can also have name same as class name, but still they have return type, though which we can identify them that they are methods not constructors.**
6. If you don’t implement any constructor within the class, compiler will do it for.
7. **this() and super() should be the first statement in the constructor code.** If you don’t mention them, compiler does it for you accordingly.
8. Constructor overloading is possible but overriding is not possible. Which means we can have overloaded constructor in our class but we can’t override a constructor.
9. Constructors can not be inherited.
10. If Super class doesn’t have a no-arg(default) constructor then compiler would not insert a default constructor in child class as it does in normal scenario.
11. Interfaces [do not have constructors](https://beginnersbook.com/2013/12/java-constructor-in-interface/).
12. Abstract class can have constructor and it gets invoked when a class, which implements interface, is instantiated. (i.e. object creation of concrete class).
13. A constructor can also invoke another constructor of the same class – By using this(). If you want to invoke a parameterized constructor then do it like this: **this(parameter list)**.

Bitset Class

The BitSet class creates a special type of array that holds bit values. The BitSet array can increase in size as needed. This makes it similar to a vector of bits. This is a legacy class but it has been completely re-engineered in Java 2, version 1.4.

The BitSet defines the following two constructors.

|  |  |
| --- | --- |
| **Sr.No.** | **Constructor & Description** |
| 1 | **BitSet( )**  This constructor creates a default object. |
| 2 | **BitSet(int size)**  This constructor allows you to specify its initial size, i.e., the number of bits that it can hold. All bits are initialized to zero. |

he following program illustrates several of the methods supported by this data structure −

[Live Demo](http://tpcg.io/TeRT19)

import java.util.BitSet;

public class BitSetDemo {

public static void main(String args[]) {

BitSet bits1 = new BitSet(16);

BitSet bits2 = new BitSet(16);

// set some bits

for(int i = 0; i < 16; i++) {

if((i % 2) == 0) bits1.set(i);

if((i % 5) != 0) bits2.set(i);

}

System.out.println("Initial pattern in bits1: ");

System.out.println(bits1);

System.out.println("\nInitial pattern in bits2: ");

System.out.println(bits2);

// AND bits

bits2.and(bits1);

System.out.println("\nbits2 AND bits1: ");

System.out.println(bits2);

// OR bits

bits2.or(bits1);

System.out.println("\nbits2 OR bits1: ");

System.out.println(bits2);

// XOR bits

bits2.xor(bits1);

System.out.println("\nbits2 XOR bits1: ");

System.out.println(bits2);

}

}

This will produce the following result −

Output

Initial pattern in bits1:

{0, 2, 4, 6, 8, 10, 12, 14}

Initial pattern in bits2:

{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}

bits2 AND bits1:

{2, 4, 6, 8, 12, 14}

bits2 OR bits1:

{0, 2, 4, 6, 8, 10, 12, 14}

bits2 XOR bits1:

{}

# Super keyword in java with example

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

The super keyword refers to the objects of immediate parent class. Before learning super keyword you must have the knowledge of [inheritance in Java](https://beginnersbook.com/2013/03/inheritance-in-java/) so that you can understand the examples given in this guide.

## The use of super keyword

1) To access the data members of parent class when both parent and child class have member with same name  
2) To explicitly call the no-arg and parameterized constructor of parent class  
3) To access the method of parent class when child class has overridden that method.

lass Superclass

{

int num = 100;

}

class Subclass extends Superclass

{

int num = 110;

void printNumber(){

/\* Note that instead of writing num we are

\* writing super.num in the print statement

\* this refers to the num variable of Superclass

\*/

System.out.println(super.num);

}

public static void main(String args[]){

Subclass obj= new Subclass();

obj.printNumber();

}

}

Output:  
100

**VECTOR**

Vector implements a dynamic array. It is similar to ArrayList, but with two differences −

* Vector is synchronized.
* Vector contains many legacy methods that are not part of the collections framework.

Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

Following is the list of constructors provided by the vector class.

|  |  |
| --- | --- |
| **Sr.No.** | **Constructor & Description** |
| 1 | **Vector( )**  This constructor creates a default vector, which has an initial size of 10. |
| 2 | **Vector(int size)**  This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size. |
| 3 | **Vector(int size, int incr)**  This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward. |
| 4 | **Vector(Collection c)**  This constructor creates a vector that contains the elements of collection c. |

Example

The following program illustrates several of the methods supported by this collection −

[Live Demo](http://tpcg.io/SXCPSN)

import java.util.\*;

public class VectorDemo {

public static void main(String args[]) {

// initial size is 3, increment is 2

Vector v = new Vector(3, 2);

System.out.println("Initial size: " + v.size());

System.out.println("Initial capacity: " + v.capacity());

v.addElement(new Integer(1));

v.addElement(new Integer(2));

v.addElement(new Integer(3));

v.addElement(new Integer(4));

System.out.println("Capacity after four additions: " + v.capacity());

v.addElement(new Double(5.45));

System.out.println("Current capacity: " + v.capacity());

v.addElement(new Double(6.08));

v.addElement(new Integer(7));

System.out.println("Current capacity: " + v.capacity());

v.addElement(new Float(9.4));

v.addElement(new Integer(10));

System.out.println("Current capacity: " + v.capacity());

v.addElement(new Integer(11));

v.addElement(new Integer(12));

System.out.println("First element: " + (Integer)v.firstElement());

System.out.println("Last element: " + (Integer)v.lastElement());

if(v.contains(new Integer(3)))

System.out.println("Vector contains 3.");

// enumerate the elements in the vector.

Enumeration vEnum = v.elements();

System.out.println("\nElements in vector:");

while(vEnum.hasMoreElements())

System.out.print(vEnum.nextElement() + " ");

System.out.println();

}

}

This will produce the following result −

Output

Initial size: 0

Initial capacity: 3

Capacity after four additions: 5

Current capacity: 5

Current capacity: 7

Current capacity: 9

First element: 1

Last element: 12

Vector contains 3.

Elements in vector:

1 2 3 4 5.45 6.08 7 9.4 10 11 12

**HASHTABLE**

Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.

However, Java 2 re-engineered Hashtable so that it also implements the Map interface. Thus, Hashtable is now integrated into the collections framework. It is similar to HashMap, but is synchronized.

Like HashMap, Hashtable stores key/value pairs in a hash table. When using a Hashtable, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

ollowing is the list of constructors provided by the HashTable class.

|  |  |
| --- | --- |
| **Sr.No** | **Constructor & Description** |
| 1 | **Hashtable( )**  This is the default constructor of the hash table it instantiates the Hashtable class. |
| 2 | **Hashtable(int size)**  This constructor accepts an integer parameter and creates a hash table that has an initial size specified by integer value size. |
| 3 | **Hashtable(int size, float fillRatio)**  This creates a hash table that has an initial size specified by size and a fill ratio specified by fillRatio. This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward. |
| 4 | **Hashtable(Map < ? extends K, ? extends V > t)**  This constructs a Hashtable with the given mappings. |

Example

The following program illustrates several of the methods supported by this data structure −

[Live Demo](http://tpcg.io/5eD0B4)

import java.util.\*;

public class HashTableDemo {

public static void main(String args[]) {

// Create a hash map

Hashtable balance = new Hashtable();

Enumeration names;

String str;

double bal;

balance.put("Zara", new Double(3434.34));

balance.put("Mahnaz", new Double(123.22));

balance.put("Ayan", new Double(1378.00));

balance.put("Daisy", new Double(99.22));

balance.put("Qadir", new Double(-19.08));

// Show all balances in hash table.

names = balance.keys();

while(names.hasMoreElements()) {

str = (String) names.nextElement();

System.out.println(str + ": " + balance.get(str));

}

System.out.println();

// Deposit 1,000 into Zara's account

bal = ((Double)balance.get("Zara")).doubleValue();

balance.put("Zara", new Double(bal + 1000));

System.out.println("Zara's new balance: " + balance.get("Zara"));

}

}

This will produce the following result −

Output

Qadir: -19.08

Zara: 3434.34

Mahnaz: 123.22

Daisy: 99.22

Ayan: 1378.0

Zara's new balance: 4434.34

**PROPERTIES**

Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.

The Properties class is used by many other Java classes. For example, it is the type of object returned by System.getProperties( ) when obtaining environmental values.

Properties define the following instance variable. This variable holds a default property list associated with a Properties object.

Properties defaults;

Following is the list of constructors provided by the properties class.

|  |  |
| --- | --- |
| **Sr.No.** | **Constructor & Description** |
| 1 | **Properties( )**  This constructor creates a Properties object that has no default values. |
| 2 | **Properties(Properties propDefault)**  Creates an object that uses propDefault for its default values. In both cases, the property list is empty. |

Example

The following program illustrates several of the methods supported by this data structure −

[Live Demo](http://tpcg.io/hN9PAR)

import java.util.\*;

public class PropDemo {

public static void main(String args[]) {

Properties capitals = new Properties();

Set states;

String str;

capitals.put("Illinois", "Springfield");

capitals.put("Missouri", "Jefferson City");

capitals.put("Washington", "Olympia");

capitals.put("California", "Sacramento");

capitals.put("Indiana", "Indianapolis");

// Show all states and capitals in hashtable.

states = capitals.keySet(); // get set-view of keys

Iterator itr = states.iterator();

while(itr.hasNext()) {

str = (String) itr.next();

System.out.println("The capital of " + str + " is " +

capitals.getProperty(str) + ".");

}

System.out.println();

// look for state not in list -- specify default

str = capitals.getProperty("Florida", "Not Found");

System.out.println("The capital of Florida is " + str + ".");

}

}

This will produce the following result −

Output

The capital of Missouri is Jefferson City.

The capital of Illinois is Springfield.

The capital of Indiana is Indianapolis.

The capital of California is Sacramento.

The capital of Washington is Olympia.

The capital of Florida is Not Found.

**Java Set**

## Featured snippet from the web

The **set** is an interface available in the **java**.

The **set** interface extends the Collection interface. An unordered collection or list in which duplicates are not allowed is referred to as a collection interface. ... The **set** interface use collection interface's methods to avoid the insertion of the same elements.

**DICTIONARY**

Dictionary is an abstract class that represents a key/value storage repository and operates much like Map.

Given a key and value, you can store the value in a Dictionary object. Once the value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs.

The Dictionary class has been marked as obsolete, so it's best not to use it.

The Dictionary class is obsolete. You should implement the [Map interface](https://www.tutorialspoint.com/java/java_map_interface.htm) to obtain key/value storage functionality.

## Creating a dictionary

The first step to creating a dictionary in Java is to choose a class that implements a “key-value pair” interface; a few examples include HashTables, HashMap, and LinkedHashMap.

## Syntax

The declaration and initialization of a dictionary follows the syntax below:

## Code

The code snippet below illustrates how to create a dictionary in Java and the usage of a few of the class’s methods:

import java.util.\*;

class My\_Dictionary

{

    public static void main(String[] args)

    {

        // creating a My HashTable Dictionary

        Hashtable<String, String> my\_dict = new Hashtable<String, String>();

    // Using a few dictionary Class methods

        // using put method

        my\_dict.put("01", "Apple");

        my\_dict.put("10", "Banana");

        // using get() method

    System.out.println("\nValue at key = 10 : " + my\_dict.get("10"));

    System.out.println("Value at key = 11 : " + my\_dict.get("11"));

        // using isEmpty() method

    System.out.println("\nIs my dictionary empty? : " + my\_dict.isEmpty() + "\n");

        // using remove() method

        // remove value at key 10

    my\_dict.remove("10");

    System.out.println("Checking if the removed value exists: " + my\_dict.get("10"));

    System.out.println("\nSize of my\_dict : " + my\_dict.size());

    }

}

**STACK**

Stack is a subclass of Vector that implements a standard last-in, first-out stack.

Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.

The following program illustrates several of the methods supported by this collection −

[Live Demo](http://tpcg.io/iIs6mr)

import java.util.\*;

public class StackDemo {

static void showpush(Stack st, int a) {

st.push(new Integer(a));

System.out.println("push(" + a + ")");

System.out.println("stack: " + st);

}

static void showpop(Stack st) {

System.out.print("pop -> ");

Integer a = (Integer) st.pop();

System.out.println(a);

System.out.println("stack: " + st);

}

public static void main(String args[]) {

Stack st = new Stack();

System.out.println("stack: " + st);

showpush(st, 42);

showpush(st, 66);

showpush(st, 99);

showpop(st);

showpop(st);

showpop(st);

try {

showpop(st);

} catch (EmptyStackException e) {

System.out.println("empty stack");

}

}

}

This will produce the following result −

## Output

stack: [ ]

push(42)

stack: [42]

push(66)

stack: [42, 66]

push(99)

stack: [42, 66, 99]

pop -> 99

stack: [42, 66]

pop -> 66

stack: [42]

pop -> 42

stack: [ ]

pop -> empty stack

**class** Learning1

{

// Pushing element on the top of the stack

**static** **void** stack\_push(Stack<Integer> stack)

{

System.***out***.println("Push Operation:");

**for**(**int** i = 0; i < 5; i++)

{

stack.push(i);

System.***out***.println(i);

System.***out***.println("Size of Stack "+stack.size());

}

}

// Popping element from the top of the stack

**static** **void** stack\_pop(Stack<Integer> stack)

{

System.***out***.println("Pop Operation:");

**for**(**int** i = 0; i < 5; i++)

{

Integer y = (Integer) stack.pop();

System.***out***.println(y);

System.***out***.println("Size of Stack "+stack.size());

}

}

// Displaying element on the top of the stack

**static** **void** stack\_peek(Stack<Integer> stack)

{

Integer element = (Integer) stack.peek();

System.***out***.println("Element on stack top: " + element);

}

// Searching element in the stack

**static** **void** stack\_search(Stack<Integer> stack, **int** element)

{

Integer pos = (Integer) stack.search(element);

**if**(pos == -1)

System.***out***.println("Element not found");

**else**

System.***out***.println("Element is found at position: " + pos);

}

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

{

Stack<Integer> stack = **new** Stack<Integer>();

*stack\_push*(stack);

*stack\_pop*(stack);

*stack\_push*(stack);

*stack\_peek*(stack);

*stack\_search*(stack, 2);

*stack\_search*(stack, 6);

}

}

**COLLECTIONS**

The standard collection classes are summarized in the following table −

|  |  |
| --- | --- |
| **Sr.No.** | **Class & Description** |
| 1 | **AbstractCollection**  Implements most of the Collection interface. |
| 2 | **AbstractList**  Extends AbstractCollection and implements most of the List interface. |
| 3 | **AbstractSequentialList**  Extends AbstractList for use by a collection that uses sequential rather than random access of its elements. |
| 4 | [LinkedList](https://www.tutorialspoint.com/java/java_linkedlist_class.htm)  Implements a linked list by extending AbstractSequentialList. |
| 5 | [ArrayList](https://www.tutorialspoint.com/java/java_arraylist_class.htm)  Implements a dynamic array by extending AbstractList. |
| 6 | **AbstractSet**  Extends AbstractCollection and implements most of the Set interface. |
| 7 | [HashSet](https://www.tutorialspoint.com/java/java_hashset_class.htm)  Extends AbstractSet for use with a hash table. |
| 8 | [LinkedHashSet](https://www.tutorialspoint.com/java/java_linkedhashset_class.htm)  Extends HashSet to allow insertion-order iterations. |
| 9 | [TreeSet](https://www.tutorialspoint.com/java/java_treeset_class.htm)  Implements a set stored in a tree. Extends AbstractSet. |
| 10 | **AbstractMap**  Implements most of the Map interface. |
| 11 | [HashMap](https://www.tutorialspoint.com/java/java_hashmap_class.htm)  Extends AbstractMap to use a hash table. |
| 12 | [TreeMap](https://www.tutorialspoint.com/java/java_treemap_class.htm)  Extends AbstractMap to use a tree. |
| 13 | [WeakHashMap](https://www.tutorialspoint.com/java/java_weakhashmap_class.htm)  Extends AbstractMap to use a hash table with weak keys. |
| 14 | [LinkedHashMap](https://www.tutorialspoint.com/java/java_linkedhashmap_class.htm)  Extends HashMap to allow insertion-order iterations. |
| 15 | [IdentityHashMap](https://www.tutorialspoint.com/java/java_identityhashmap_class.htm)  Extends AbstractMap and uses reference equality when comparing documents. |

The *AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList* and *AbstractMap* classes provide skeletal implementations of the core collection interfaces, to minimize the effort required to implement them.

The following legacy classes defined by java.util have been discussed in the previous chapter −

|  |  |
| --- | --- |
| **Sr.No.** | **Class & Description** |
| 1 | [Vector](https://www.tutorialspoint.com/java/java_vector_class.htm)  This implements a dynamic array. It is similar to ArrayList, but with some differences. |
| 2 | [Stack](https://www.tutorialspoint.com/java/java_stack_class.htm)  Stack is a subclass of Vector that implements a standard last-in, first-out stack. |
| 3 | [Dictionary](https://www.tutorialspoint.com/java/java_dictionary_class.htm)  Dictionary is an abstract class that represents a key/value storage repository and operates much like Map. |
| 4 | [Hashtable](https://www.tutorialspoint.com/java/java_hashtable_class.htm)  Hashtable was part of the original java.util and is a concrete implementation of a Dictionary. |
| 5 | [Properties](https://www.tutorialspoint.com/java/java_properties_class.htm)  Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String. |
| 6 | [BitSet](https://www.tutorialspoint.com/java/java_bitset_class.htm)  A BitSet class creates a special type of array that holds bit values. This array can increase in size as needed. |

**TREEMAP**

The TreeMap class implements the Map interface by using a tree. A TreeMap provides an efficient means of storing key/value pairs in sorted order, and allows rapid retrieval.

You should note that, unlike a hash map, a tree map guarantees that its elements will be sorted in an ascending key order.

Following is the list of the constructors supported by the TreeMap class.

|  |  |
| --- | --- |
| **Sr.No.** | **Constructors & Description** |
| 1 | **TreeMap( )**  This constructor constructs an empty tree map that will be sorted using the natural order of its keys. |
| 2 | **TreeMap(Comparator comp)**  This constructor constructs an empty tree-based map that will be sorted using the Comparator comp. |
| 3 | **TreeMap(Map m)**  This constructor initializes a tree map with the entries from **m**, which will be sorted using the natural order of the keys. |
| 4 | **TreeMap(SortedMap sm)**  This constructor initializes a tree map with the entries from the SortedMap **sm**, which will be sorted in the same order as **sm**. |

What is the difference between set list and map?

**Set** and all of the classes which implements **Set** interface should have unique elements. **Map** stored the elements as key & value pair. **Map** doesn't allow duplicate keys while it allows duplicate values. 2) Null values: **List** allows any number of null values.

The **difference** is that **Set** is stored **in** unordered way and does not allow duplicate values. **List** is used to store elements **in** ordered way and it does allow duplicate values. **Set** elements cannot be accessed by an index position, and **List** elements can be accessed with an index position.Jul 19

Which is faster HashSet or HashMap?

**HashMap** is **faster**/ than **HashSet** because values are associated with a unique key. **HashSet** is slower than **HashMap** because the member object is used for calculating hashcode value, which can be same for two objects.

HashSet and ArrayList both are some of the most important classes of the Java Collection framework.

The following are the important differences between ArrayList and HashSet.

| **Sr. No.** | **Key** | **ArrayList** | **HashSet** |
| --- | --- | --- | --- |
| 1 | Implementation | ArrayList is the implementation of the list interface. | HashSet on the other hand is the implementation of a set interface. |
| 2 | Internal implementation | ArrayList internally implements array for its implementation. | HashSet internally uses Hashmap for its implementation. |
| 3 | Order of elements | ArrayList maintains the insertion order i.e order of the object in which they are inserted. | HashSet is an unordered collection and doesn't maintain any order. |
| 4 | Duplicates | ArrayList allows duplicate values in its collection. | On other hand duplicate elements are not allowed in Hashset. |
| 5 | Index performance | ArrayList uses index for its performance i.e its index based one can retrieve object by calling get(index) or remove objects by calling remove(index) | HashSet is completely based on object also it doesn't provide get() method. |
| 6 | Null Allowed | Any number of null value can be inserted in arraylist without any restriction. | On other hand Hashset allows only one null value in its collection,after which no null value is allowed to be added. |

System.***out***.println("Array list Example");

List<Integer> tp = **new** ArrayList<>(5);

tp.add(23);

tp.add(23);

tp.add(25);

tp.add(28);

tp.add(23);

**for**(Integer s:tp)

{

System.***out***.println(s);

}

System.***out***.println("Hashset list Example");

Set<Integer> st = **new** HashSet<>(5);

st.add(53);

st.add(43);

st.add(73);

st.add(23);

st.add(53);

**for**(Integer si:st)

{

System.***out***.println(si);

}

Array list Example

23

23

25

28

23

Hashset list Example

73

43

53

23

**ArrayList** provides constant time for search operation, so it is better to **use ArrayList** if searching is more frequent operation than add and remove operation. The **LinkedList** provides constant time for add and remove operations. So it is better to **use LinkedList** for manipulation.

**SORTED SET**

The SortedSet interface extends Set and declares the behavior of a set sorted in an ascending order. In addition to those methods defined by Set, the SortedSet interface declares the methods summarized in the following table −

Several methods throw a NoSuchElementException when no items are contained in the invoking set. A ClassCastException is thrown when an object is incompatible with the elements in a set.

A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the set.

import java.util.\*;

public class AlgorithmsDemo {

public static void main(String args[]) {

// Create and initialize linked list

LinkedList ll = new LinkedList();

ll.add(new Integer(-8));

ll.add(new Integer(20));

ll.add(new Integer(-20));

ll.add(new Integer(8));

// Create a reverse order comparator

Comparator r = Collections.reverseOrder();

// Sort list by using the comparator

Collections.sort(ll, r);

// Get iterator

Iterator li = ll.iterator();

System.out.print("List sorted in reverse: ");

while(li.hasNext()) {

System.out.print(li.next() + " ");

}

System.out.println();

Collections.shuffle(ll);

// display randomized list

li = ll.iterator();

System.out.print("List shuffled: ");

while(li.hasNext()) {

System.out.print(li.next() + " ");

}

System.out.println();

System.out.println("Minimum: " + Collections.min(ll));

System.out.println("Maximum: " + Collections.max(ll));

}

}

This will produce the following result −

## Output

List sorted in reverse: 20 8 -8 -20

List shuffled: 20 -20 8 -8

Minimum: -20

Maximum: 20

## The Collection Interfaces

The collections framework defines several interfaces. This section provides an overview of each interface −

|  |  |
| --- | --- |
| **Sr.No.** | **Interface & Description** |
| 1 | [The Collection Interface](https://www.tutorialspoint.com/java/java_collection_interface.htm)  This enables you to work with groups of objects; it is at the top of the collections hierarchy. |
| 2 | [The List Interface](https://www.tutorialspoint.com/java/java_list_interface.htm)  This extends **Collection** and an instance of List stores an ordered collection of elements. |
| 3 | [The Set](https://www.tutorialspoint.com/java/java_set_interface.htm)  This extends Collection to handle sets, which must contain unique elements. |
| 4 | [The SortedSet](https://www.tutorialspoint.com/java/java_sortedset_interface.htm)  This extends Set to handle sorted sets. |
| 5 | [The Map](https://www.tutorialspoint.com/java/java_map_interface.htm)  This maps unique keys to values. |
| 6 | [The Map.Entry](https://www.tutorialspoint.com/java/java_mapentry_interface.htm)  This describes an element (a key/value pair) in a map. This is an inner class of Map. |
| 7 | [The SortedMap](https://www.tutorialspoint.com/java/java_sortedmap_interface.htm)  This extends Map so that the keys are maintained in an ascending order. |
| 8 | [The Enumeration](https://www.tutorialspoint.com/java/java_enumeration_interface.htm)  This is legacy interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superceded by Iterator. |

It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array, or an array of any type that supports ordering.

Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

## Generic Methods

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods −

* All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type ( < E > in the next example).
* Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
* The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
* A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

public class GenericMethodTest {

// generic method printArray

public static < E > void printArray( E[] inputArray ) {

// Display array elements

for(E element : inputArray) {

System.out.printf("%s ", element);

}

System.out.println();

}

public static void main(String args[]) {

// Create arrays of Integer, Double and Character

Integer[] intArray = { 1, 2, 3, 4, 5 };

Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };

Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

System.out.println("Array integerArray contains:");

printArray(intArray); // pass an Integer array

System.out.println("\nArray doubleArray contains:");

printArray(doubleArray); // pass a Double array

System.out.println("\nArray characterArray contains:");

printArray(charArray); // pass a Character array

}

}

This will produce the following result −

### **Output**

Array integerArray contains:

1 2 3 4 5

Array doubleArray contains:

1.1 2.2 3.3 4.4

Array characterArray contains:

H E L L O

**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.
2. 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.
3. Generics promotes code reusability.
4. 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.

**Generics** means **parameterized types**. The idea is to allow type (Integer, String, … etc, and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types.

An entity such as class, interface, or method that operates on a parameterized type is called generic entity.

**Why Generics?**  
**Object** is the superclass of all other classes and Object reference can refer to any type object. These features lack type safety. Generics adds that type safety feature. We will discuss that type safety feature in later examples.

## Generic Methods

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods −

* All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type ( < E > in the next example).

Serialization

Well, **serialization** allows us to convert the state of an object into a byte stream, which then can be saved into a file on the local disk or sent over the network to any other machine. And deserialization allows us to reverse the process, which means reconverting the **serialized** byte stream to an object again

## What Are Serialization and Deserialization?

In Java, we create several objects that live and die accordingly, and every object will certainly die when the JVM dies. But sometimes, we might want to reuse an object between several JVMs or we might want to transfer an object to another machine over the network.

Well, **serialization** allows us to convert the state of an object into a byte stream, which then can be saved into a file on the local disk or sent over the network to any other machine. And **deserialization** allows us to reverse the process, which means reconverting the serialized byte stream to an object again.

In simple words, object **serialization** is the process of saving an object's state to a sequence of bytes and **deserialization** is the process of reconstructing an object from those bytes. Generally, the complete process is called **serialization,** but I think it is better to classify both as separate for more clarity:

## Serializing an Object

The ObjectOutputStream class is used to serialize an Object. The following SerializeDemo program instantiates an Employee object and serializes it to a file.

When the program is done executing, a file named employee.ser is created. The program does not generate any output, but study the code and try to determine what the program is doing.

**Note** − When serializing an object to a file, the standard convention in Java is to give the file a **.ser** extension.

### **Example**

import java.io.\*;

public class SerializeDemo {

public static void main(String [] args) {

Employee e = new Employee();

e.name = "Reyan Ali";

e.address = "Phokka Kuan, Ambehta Peer";

e.SSN = 11122333;

e.number = 101;

try {

FileOutputStream fileOut =

new FileOutputStream("/tmp/employee.ser");

ObjectOutputStream out = new ObjectOutputStream(fileOut);

out.writeObject(e);

out.close();

fileOut.close();

System.out.printf("Serialized data is saved in /tmp/employee.ser");

} catch (IOException i) {

i.printStackTrace();

}

}

}

## Deserializing an Object

The following DeserializeDemo program deserializes the Employee object created in the SerializeDemo program. Study the program and try to determine its output −

### **Example**

import java.io.\*;

public class DeserializeDemo {

public static void main(String [] args) {

Employee e = null;

try {

FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");

ObjectInputStream in = new ObjectInputStream(fileIn);

e = (Employee) in.readObject();

in.close();

fileIn.close();

} catch (IOException i) {

i.printStackTrace();

return;

} catch (ClassNotFoundException c) {

System.out.println("Employee class not found");

c.printStackTrace();

return;

}

System.out.println("Deserialized Employee...");

System.out.println("Name: " + e.name);

System.out.println("Address: " + e.address);

System.out.println("SSN: " + e.SSN);

System.out.println("Number: " + e.number);

}

}

This will produce the following result −

### **Output**

Deserialized Employee...

Name: Reyan Ali

Address:Phokka Kuan, Ambehta Peer

SSN: 0

Number:101

The **transient** keyword in **Java** is used to avoid serialization. If any object of a data structure is defined as a **transient**, then it will not be serialized. Serialization is the ​process of converting an object into a byte stream.

Java access modifiers are used to provide access control in java. Java provides access control through three keywords – **private**, **protected** and public. We are not required to use these access modifiers always, so we have another one namely “**default** access“, “package-**private**” or “no modifier“.

Can constructor be private?

Yes, we **can** declare a **constructor** as **private**. If we declare a **constructor** as **private** we are not able to create an object of a class. We **can** use this **private constructor** in the Singleton Design Pattern

Can we override static method?

**Can we override** a **static method**? No, **we** cannot **override static methods** because **method overriding** is based on dynamic binding at runtime and the **static methods** are bonded using **static** binding at compile time. So, **we** cannot **override static methods**.