**1.Introduction, Roles and responsibilities:**

**A:** **SDET (Software Development Engineer in Test)** in testing is an IT professional who can work equally and effectively in both development and testing roles. SDETs take part in complete software development process as well as software testing process. SDET professional’s knowledge is entirely focused on testability, robustness and performance of software testing and development process.

**Difference between SDET and Tester?**

|  |  |
| --- | --- |
| **SDET** | **Manual Tester** |
| Knows the entire system start to end | Limited knowledge about the system |
| SDET is involved in every step of the software development process like  Designing, development, and testing. | QA is only involved in the testing life cycle of the software development process. |
| Highly skilled professional with development as well as testing knowledge. | Software tester is only involved in preparing and executing the test cases |
| SDET can participate in test automation tool development and may make it for generic use. | Not expected to develop test automation tools or frameworks. |
| SDETs need to perform duties like performance testing, automated generation of test data, etc. | Only testing related task will be performed by the tester. |
| Know requirements and guidelines for the products | No such knowledge expected from QA professionals. |

## When do you need SDET?

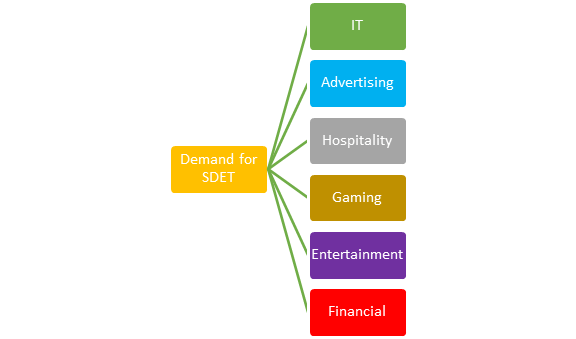
Today organizations are looking for a professional who can take part in software development. At the same time, he should also handle testing of the developed software. That’s why hiring SDET helps them as they can work for developing high-performance code or designing the testing framework.

**Benefits of SDET professional:**

* SDETs professionals can automate the acceptance test by leveraging their coding skills
* These professionals deal with end users or customers and their experiences. They already possessed engineering skills so they can also think like a developer.
* Extensive code coverage through unit testing
* Can build, deploy, run & manage the application individually
* Influence the development, program management, & design teams on technical implementation and user scenarios

## Industries in need of SDET

There is a trend among companies to adopt new technology and methodologies. Therefore, SDET software professionals are highly on demand. Software developer engineer in the test are mainly needed in the following fields:



**What are the roles and responsibilities of an SDET?**

* SDET should able to perform Test Automation and setting up frameworks on multiple application platforms like Web, Mobile, and Desktop.
* Investigate customer problems referred by the technical support team.
* Create & manage bug reports and communicate with the team.
* Able to build different test scenarios and acceptance tests.
* SDET needs to handle technical communications with Partners to understand client’s systems or APIs.
* SDET also work with deployments teams and resolving any level issues for the system.
* SDET should also able to set up, maintain, and operate test automation frameworks.

**2.Explain OOPS concepts in JAVA:**

**A:What is Object-Oriented Programming?**

**Object-oriented programming** has a sweeping impact because it appeals at multiple levels and promises faster and cheaper development and maintenance. It follows a bottom-up approach to develop applications.

In this section, we will discuss in-depth **what is object-oriented programming?**

## Object-Oriented Programming

The word **object-oriented** is the combination of two words i.e. **object** and **oriented**. The dictionary meaning of the object is an article or entity that exists in the real world. The meaning of oriented is interested in a particular kind of thing or entity. In layman's terms, it is a programming pattern that rounds around an object or entity are called **object-oriented programming.**



The technical definition of object-oriented programming is as follows:

The **object-oriented programming** is basically a computer programming design philosophy or methodology that organizes/ models software design around data, or objects rather than functions and logic.

An object is referred to as a data field that has unique attributes and behavior. Everything in OOP is grouped as self-sustainable objects.

It is the most popular programming model among developers. It is well suited for programs that are large, complex, and actively updated or maintained. It simplifies software development and maintenance by providing major concepts such as **abstraction, inheritance, polymorphism**, and **encapsulation**. These core concepts support OOP.

A real-world example of OOP is the automobile. It more completely illustrates the power of object-oriented design.

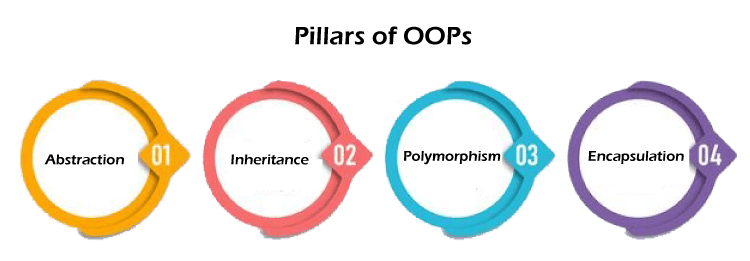
### **Points to Remember**

* Everything is an object
* Developer manipulates objects that uses message passing.
* Every object is an instance of a class.
* The class contains the attribute and behavior associated with an object.

## Pillars of OOPs

The major concepts that we have discussed above are known as **pillars of OOPs**. There are **four** pillars on which OOP rests.

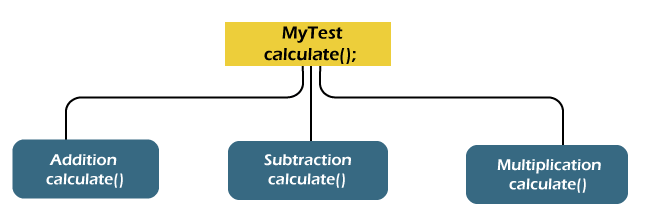
* Abstraction
* Encapsulation
* Inheritance
* Polymorphism



Let's discuss each in detail.

### **Abstraction**

The concept allows us to hide the implementation from the user but shows only essential information to the user. Using the concept developer can easily make changes and added over time.

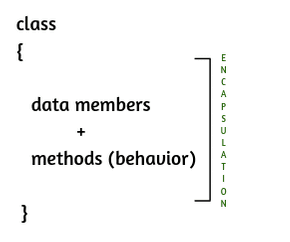


There are the following advantages of abstraction:

* It reduces complexity.
* It avoids delicacy.
* Eases the burden of maintenance
* Increase security and confidentially.

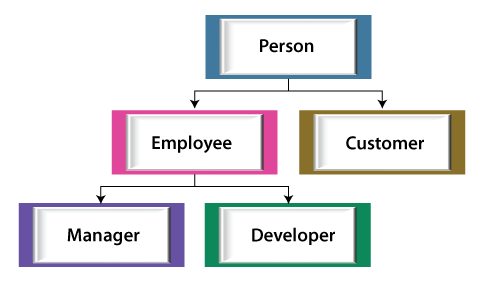
### **Encapsulation**

[Encapsulation](https://www.javatpoint.com/encapsulation) is a mechanism that allows us to bind data and functions of a class into an entity. It protects data and functions from outside interference and misuse. Therefore, it also provides security. A class is the best example of encapsulation.



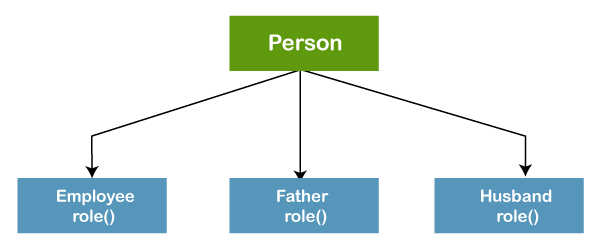
### **Inheritance**

The concept allows us to inherit or acquire the properties of an existing class (parent class) into a newly created class (child class). It is known as [**inheritance**](https://www.javatpoint.com/inheritance-in-java). It provides code reusability.



### **Polymorphism**

The word [**polymorphism**](https://www.javatpoint.com/runtime-polymorphism-in-java) is derived from the two words i.e. **ploy** and **morphs**. Poly means many and morphs means forms. It allows us to create methods with the same name but different method signatures. It allows the developer to create clean, sensible, readable, and resilient code.

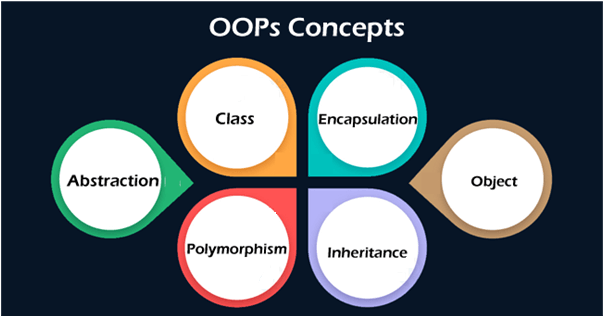


The above figure best describes the concepts of polymorphism. A person plays an employee role in the office, father and husband role in the home.

## OOPs Concepts

The [OOPs concepts](https://www.javatpoint.com/java-oops-concepts) include the following:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation



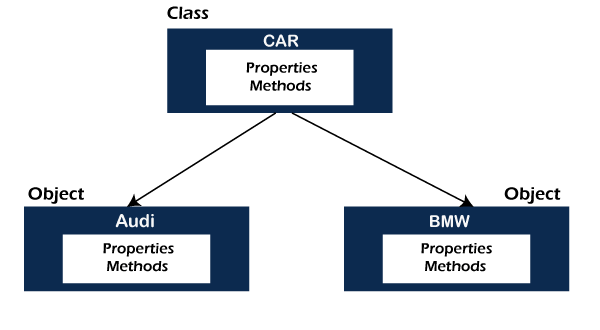
### **Object**

An [object](https://www.javatpoint.com/object-and-class-in-java#object) is a real-world entity that has attributes, behavior, and properties. It is referred to as an instance of the class. It contains member functions, variables that we have defined in the class. It occupies space in the memory. Different objects have different states or attributes, and behaviours.

### **Class**

A [class](https://www.javatpoint.com/object-and-class-in-java#class) is a blueprint or template of an object. It is a user-defined data type. Inside a class, we define variables, constants, member functions, and other functionality. it binds data and functions together in a single unit. It does not consume memory at run time. Note that classes are not considered as a data structure. It is a logical entity. It is the best example of data binding. Note that a class can exist without an object but vice-versa is not possible.

The following figure best illustrates the class and object in OOP.



Apart from these core concepts, there are some other object-oriented concepts used in OOP.

### **Coupling**

In programming, separation of concerns is known as **coupling**. It means that an object cannot directly change or modify the state or behavior of other objects. It defines how closely two objects are connected together. There are two types of coupling, **loose** coupling, and **tight** coupling.

Objects that are independent of one another and do not directly modify the state of other objects is called loosely coupled. Loose coupling makes the code more flexible, changeable, and easier to work with.

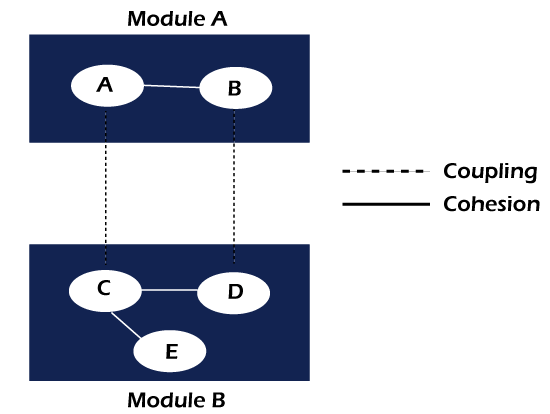
Objects that depend on other objects and can modify the states of other objects are called tightly coupled. It creates conditions where modifying the code of one object also requires changing the code of other objects. The reuse of code is difficult in tight coupling because we cannot separate the code.

Since using loose coupling is always a good habit.

### **Cohesion**

In OOP, **cohesion** refers to the degree to which the elements inside a module belong together. It measures the strength of the relationship between the module and data. In short, cohesion represents the clarity of the responsibilities of a module. It is often contrasted with coupling.

It focuses on a how single module or class is intended. Higher the cohesiveness of the module or class, better is the object-oriented design.



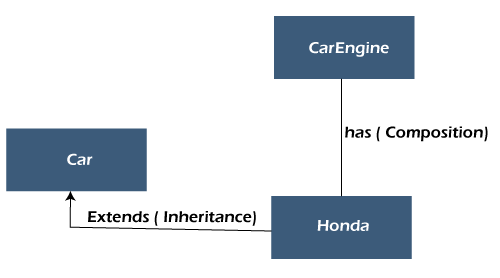
There are two types of cohesion, i.e. **High** and **Low**.

* High cohesion is associated with several required qualities of software including **robustness, reliability**, and **understandability**.
* Low cohesion is associated with unwanted qualities such as being difficult to **maintain, test, reuse,** or even **understand**.

High cohesion often associates with loose coupling and vice versa.

### **Composition**

Composition is one of the vital concepts in OOP. It describes a class that references one or more objects of other classes in instance variables. It allows us to model a has-a association between objects. We can find such relationships in the real world. For example, a car has an engine. the following figure depicts the same



The main benefits of composition are:

* Reuse existing code
* Design clean APIs
* Change the implementation of a class used in a composition without adapting any external clients.

### **Association**

The association defines the relationship between the objects. Note that an object can be associated with one or more than one object. The relationship can be unidirectional or bidirectional. There are the following types of association.

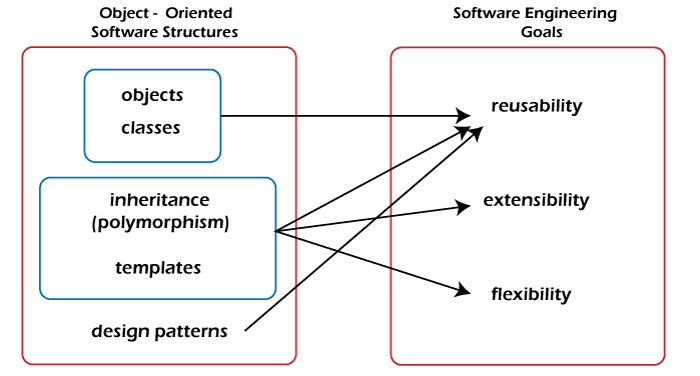
* One to One
* One to Many
* Many to One, and
* Many to Many

### **Aggregation**

It is an advanced form of association in which each object has its own Lifecycle but there exists ownership as well. In other words, a relationship where a child can exist independently of the parent. It is also termed as has-a relationship in Java. Like, inheritance represents the is-a relationship. It is another way to reuse objects.

## Why should we use OOP?

Object-oriented programming is an evolutionary development in software engineering. Using OOP in software development is a good habit because it accomplishes the three major software engineering goals, as we have shown in the following figure.



## Where it is used?

OOP is often the best use when we are dealing with manufacturing and designing applications. It provides modularity in programming. It allows us to break down the software into chunks of small problems that we then can solve one object at a time.

It should be used where the reusability of code and maintenance is a major concern. Because it makes development easy and we can easily append code without affecting other code blocks. It should be used where complex programming is a challenge.

## Benefits of OOP

* Modular, scalable, extensible, reusable, and maintainable.
* It models the complex problem in a simple structure.
* Object can be used across the program.
* Code can be reused.
* We can easily modify, append code without affecting the other code blocs.
* Provides security through encapsulation and data hiding features.
* Beneficial to collaborative development in which a large project is divided into groups.
* Debugging is easy.

## Limitations of OOP

* Requires intensive testing processes.
* Solving problems takes more time as compared to Procedure Oriented Programming.
* The size of the programs created using this approach may become larger than the programs written using the procedure-oriented programming approach.
* Software developed using this approach requires a substantial amount of pre-work and planning.
* OOP code is difficult to understand if you do not have the corresponding class documentation.
* In certain scenarios, these programs can consume a large amount of memory.
* Not suitable for small problems.
* Takes more time to solve problems

## Applications of OOPs

* Computer graphics applications
* Object-oriented database
* User-interface design such as windows
* Real-time systems
* Simulation and modeling
* Client-Server System
* Artificial Intelligence System
* CAD/CAM Software
* Office automation system

**3.Difference between i)WebDriver driver = new ChromeDriver():**

WebDriver driver = new ChromeDriver();

This means that type of driver is WebDriver type and it's not specific so you can change browser or implement cross browser automation testing.

But if you do

**ii)ChromeDriver driver = new ChromeDriver():**

ChromeDriver driver = new ChromeDriver();

You can not achieve cross browser testing.

ChromeDriver driver = new ChromeDriver()

Then only methods defined in the ChromeDriver class will be exposed. So we are upcasting it to the highest level which is the WebDriver interface so that **all methods in WebDriver interface, and overridden in RemoteWebDriver, and ChromeDriver class are available in the object instance**.

This help us to use the same variable to initialize different types of drivers as only methods implemented/defined by the web driver class is used.

WebDriver driver = new ChromeDriver()

WebDriver driver = new FireFoxDriver()

**iii)Test test = new Test():**

Test t1=new Test();

Here, you are creating an instance of Test class and assigned it to t1

Test t2=new Test(){ };

Here, You have created an anonymous sub class of Test and instantiate it and assigned to t2

**4.Collections framework with examples?**

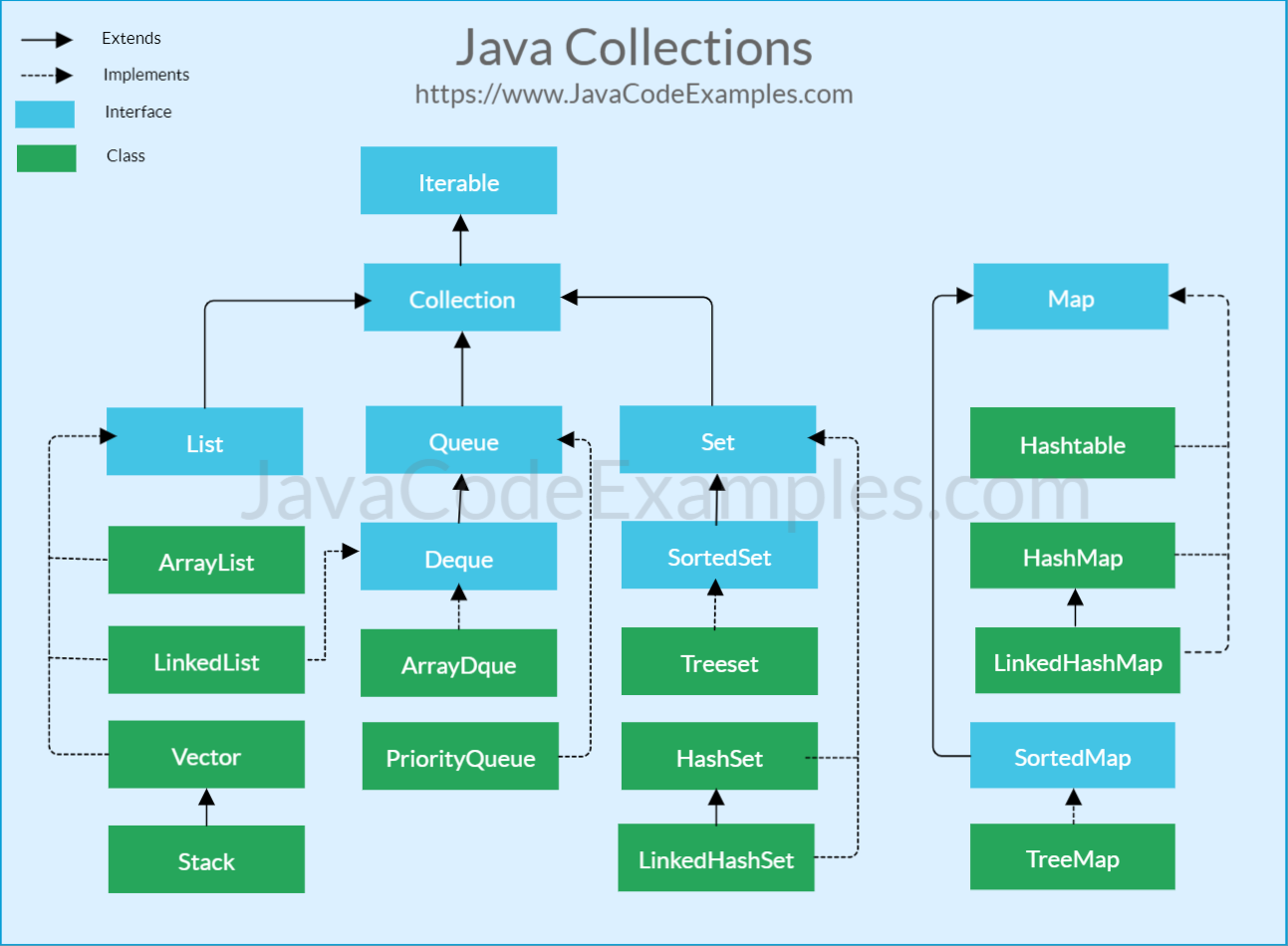
**A:**

Java Collection framework tutorial with examples will help you understand how to use the collection classes in an easier way. A collection in Java is an object that stores a group of objects as a single unit. Java collection framework provides several interfaces and classes to help group the objects as a unit.

The Java collection framework has two main interfaces, the Collection interface, and the Map interface. Several other classes and interfaces implement and extend these two main interfaces to make the collection framework.

Here is the pictorial representation of the Java Collection Framework hierarchy.

(click image to open it in a new window)

[](https://www.javacodeexamples.com/wp-content/uploads/java-collections-cheat-sheet.png)

As you can see from the above collection framework hierarchy, the Collection interface and Map interface are at the top of the hierarchy.

## Collection Interface

The Collection interface is the root interface and provides common methods like add, remove, clear, contains, equals, hash code, and iterator.

## List interface

The List interface extends the Collections interface. The List represents index based ordered collection of the objects. The elements contained in the List are ordered and can be inserted, accessed or searched based on their index. The list may contain duplicate elements. The main classes implementing the List interface are ArrayList and LinkedList.

## Set interface

The Set interface extends the Collection interface and represents a collection that does not contain any duplicate elements (it can only have one null element as well). The main classes implementing the Set interface are TreeSet, HashSet and LinkedHashSet.

## Queue Interface

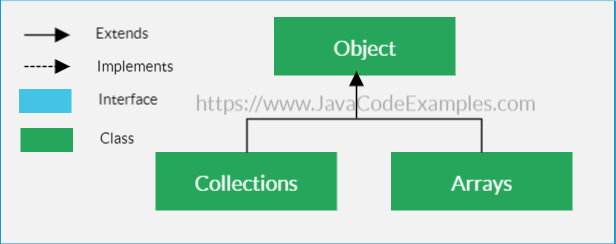
The Queue interface extends Collection interface and represents a collection that is usually ordered by FIFO (first in first out) order. The main classes implementing the Set interface are LinkedList, ArrayDeque and PriorityQueue.

## Map Interface

The map interface is a root interface and allows storing the key value pairs. The map does not allow duplicate keys. Map interface does not guarantee the order of the elements, however, some implementations like TreeMap does. The main classes implementing the Set interface are Hashtable, HashMap, TreeMap and LinkedHashMap.

## Utility Classes

In addition to the above mentioned main interfaces and classes, there are two utility classes that are part of the Java collection framework.

[](https://www.javacodeexamples.com/wp-content/uploads/Java-Collections-Arrays-1.png)

### 1. Collections class

The Collections class contains static utility methods that either accepts or returns the collection. The collection class provides many useful methods for shuffling, reversing, sorting and searching collection objects.

### 2. Arrays Class

Similar to the Collections class, the Arrays class contains static utility methods for manipulating arrays. The Arrays class provides many useful methods for sorting, searching, copying and filling the arrays.

**5.String Immutable with examples?**

**A:String class in Java is immutable**. The meaning of immutable is unchangeable or unmodifiable.

That is, once we create a [string](https://www.scientecheasy.com/2020/05/string-in-java.html/) object with value, we are not allowed to perform any changes in that object.

In other words, we cannot modify the value of the string. But if you try to change with a new value, a new string object will be created by storing a new value.

So, we cannot perform any changes with the existing string object. This non-changeable behavior is nothing but an **immutability concept** in Java.

Java implements this immutability concept to minimize the duplication of string values that tend to exist many times in any application program.

Let’s understand the immutability concept with the help of an example program.

**Program code:**

package stringPrograms;

public class ImmutabilityTest

{

public static void main(String[] args)

{

String s = "hello";

s.concat("world"); // concat() method adds string at the end.

System.out.println(s); // It will print "hello" because string is immutable object.

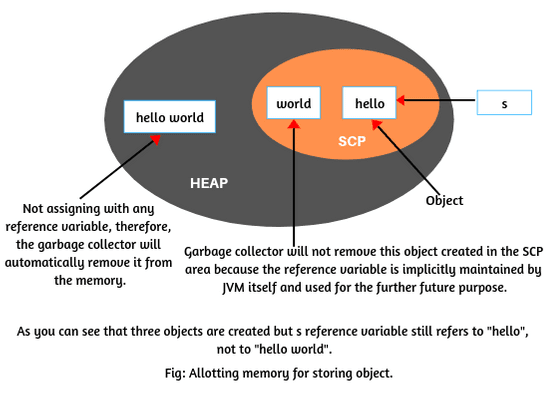
}

}

Output:

hello

Let us understand the explanation of output of the above program by the given below figure.

[](https://www.scientecheasy.com/2020/05/java-immutable-string.html/)

**Explanation:**

When JVM will execute statement String s = “hello”;, it will create a string object in the string constant pool and store “hello” in it.

When the next statement s.concat(“world”); will be executed by JVM, it will create two new objects because we are trying to modify the original content.

1. First, for every string literal “world”, JVM will create one copy of string object in the string constant pool.

2. The second object will be created in the heap with modified content “hello world”. Since string concatenation is executed at the runtime. Therefore, if a new object is required to create, this new object is always created in the heap area only, not in string constant pool.

Since this new object is not assigning with any reference variable, therefore, it is called unreferenced object and the garbage collector will automatically remove it from the memory.

Thus, the value of string s is not modified and still, ‘s’ is pointing to “hello” only. Therefore, the result is “hello”. This is the reason, string objects are called immutable in Java.

**Key point:**

1. String concatenation cannot be resolved at the compile time. It is always executed at runtime. This causes an extra object to be generated.

## Why String objects are immutable in Java?

String objects are immutable in Java because Java uses the concept of string constant pool. Suppose there are 6 reference variables, and pointing to the same object “Hello world”.

If one reference variable of them changes the value of object from “Hello world” to “Hello”, with this change, all the reference variables will be affected. That’s why string objects are immutable in Java.

The primary advantage of string immutability is that Java compiler can save space in the memory by sharing strings

**6. How to implement Hashmap?**

**A:**

Java **HashMap** class implements the Map interface which allows us to store key and value pair, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

HashMap in Java is like the legacy Hashtable class, but it is not synchronized. It allows us to store the null elements as well, but there should be only one null key. Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface.

### **Points to remember**

* Java HashMap contains values based on the key.
* Java HashMap contains only unique keys.
* Java HashMap may have one null key and multiple null values.
* Java HashMap is non synchronized.
* Java HashMap maintains no order.
* The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

### **Hierarchy of HashMap class**

As shown in the above figure, HashMap class extends AbstractMap class and implements Map interface.

### **HashMap class declaration**

Let's see the declaration for java.util.HashMap class.

1. **public** **class** HashMap<K,V> **extends** AbstractMap<K,V> **implements** Map<K,V>, Cloneable, Serializable

### **HashMap class Parameters**

Let's see the Parameters for java.util.HashMap class.

* **K**: It is the type of keys maintained by this map.
* **V**: It is the type of mapped values.

### **Constructors of Java HashMap class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| HashMap() | It is used to construct a default HashMap. |
| HashMap(Map<? extends K,? extends V> m) | It is used to initialize the hash map by using the elements of the given Map object m. |
| HashMap(int capacity) | It is used to initializes the capacity of the hash map to the given integer value, capacity. |
| HashMap(int capacity, float loadFactor) | It is used to initialize both the capacity and load factor of the hash map by using its arguments. |

### **Methods of Java HashMap class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void clear() | It is used to remove all of the mappings from this map. |
| boolean isEmpty() | It is used to return true if this map contains no key-value mappings. |
| Object clone() | It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned. |
| Set entrySet() | It is used to return a collection view of the mappings contained in this map. |
| Set keySet() | It is used to return a set view of the keys contained in this map. |
| V put(Object key, Object value) | It is used to insert an entry in the map. |
| void putAll(Map map) | It is used to insert the specified map in the map. |
| V putIfAbsent(K key, V value) | It inserts the specified value with the specified key in the map only if it is not already specified. |
| V remove(Object key) | It is used to delete an entry for the specified key. |
| boolean remove(Object key, Object value) | It removes the specified values with the associated specified keys from the map. |
| V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). |
| V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction) | It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null. |
| V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null. |
| boolean containsValue(Object value) | This method returns true if some value equal to the value exists within the map, else return false. |
| boolean containsKey(Object key) | This method returns true if some key equal to the key exists within the map, else return false. |
| boolean equals(Object o) | It is used to compare the specified Object with the Map. |
| void forEach(BiConsumer<? super K,? super V> action) | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. |
| V get(Object key) | This method returns the object that contains the value associated with the key. |
| V getOrDefault(Object key, V defaultValue) | It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key. |
| boolean isEmpty() | This method returns true if the map is empty; returns false if it contains at least one key. |
| V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction) | If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. |
| V replace(K key, V value) | It replaces the specified value for a specified key. |
| boolean replace(K key, V oldValue, V newValue) | It replaces the old value with the new value for a specified key. |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| Collection<V> values() | It returns a collection view of the values contained in the map. |
| int size() | This method returns the number of entries in the map. |

### **Java HashMap Example**

Let's see a simple example of HashMap to store key and value pair.

1. **import** java.util.\*;
2. **public** **class** HashMapExample1{
3. **public** **static** **void** main(String args[]){
4. HashMap<Integer,String> map=**new** HashMap<Integer,String>();//Creating HashMap
5. map.put(1,"Mango");  //Put elements in Map
6. map.put(2,"Apple");
7. map.put(3,"Banana");
8. map.put(4,"Grapes");
10. System.out.println("Iterating Hashmap...");
11. **for**(Map.Entry m : map.entrySet()){
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
14. }
15. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=HashMapExample1)

Iterating Hashmap...

1 Mango

2 Apple

3 Banana

4 Grapes

In this example, we are storing Integer as the key and String as the value, so we are using HashMap<Integer,String> as the type. The put() method inserts the elements in the map.

To get the key and value elements, we should call the getKey() and getValue() methods. The Map.Entry interface contains the getKey() and getValue() methods. But, we should call the entrySet() method of Map interface to get the instance of Map.Entry.

### **No Duplicate Key on HashMap**

You cannot store duplicate keys in HashMap. However, if you try to store duplicate key with another value, it will replace the value.

1. **import** java.util.\*;
2. **public** **class** HashMapExample2{
3. **public** **static** **void** main(String args[]){
4. HashMap<Integer,String> map=**new** HashMap<Integer,String>();//Creating HashMap
5. map.put(1,"Mango");  //Put elements in Map
6. map.put(2,"Apple");
7. map.put(3,"Banana");
8. map.put(1,"Grapes"); //trying duplicate key
10. System.out.println("Iterating Hashmap...");
11. **for**(Map.Entry m : map.entrySet()){
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
14. }
15. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=HashMapExample2)

Iterating Hashmap...

1 Grapes

2 Apple

3 Banana

### **Java HashMap example to add() elements**

Here, we see different ways to insert elements.

1. **import** java.util.\*;
2. **class** HashMap1{
3. **public** **static** **void** main(String args[]){
4. HashMap<Integer,String> hm=**new** HashMap<Integer,String>();
5. System.out.println("Initial list of elements: "+hm);
6. hm.put(100,"Amit");
7. hm.put(101,"Vijay");
8. hm.put(102,"Rahul");
10. System.out.println("After invoking put() method ");
11. **for**(Map.Entry m:hm.entrySet()){
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
15. hm.putIfAbsent(103, "Gaurav");
16. System.out.println("After invoking putIfAbsent() method ");
17. **for**(Map.Entry m:hm.entrySet()){
18. System.out.println(m.getKey()+" "+m.getValue());
19. }
20. HashMap<Integer,String> map=**new** HashMap<Integer,String>();
21. map.put(104,"Ravi");
22. map.putAll(hm);
23. System.out.println("After invoking putAll() method ");
24. **for**(Map.Entry m:map.entrySet()){
25. System.out.println(m.getKey()+" "+m.getValue());
26. }
27. }
28. }

Initial list of elements: {}

After invoking put() method

100 Amit

101 Vijay

102 Rahul

After invoking putIfAbsent() method

100 Amit

101 Vijay

102 Rahul

103 Gaurav

After invoking putAll() method

100 Amit

101 Vijay

102 Rahul

103 Gaurav

104 Ravi

### **Java HashMap example to remove() elements**

Here, we see different ways to remove elements.

1. **import** java.util.\*;
2. **public** **class** HashMap2 {
3. **public** **static** **void** main(String args[]) {
4. HashMap<Integer,String> map=**new** HashMap<Integer,String>();
5. map.put(100,"Amit");
6. map.put(101,"Vijay");
7. map.put(102,"Rahul");
8. map.put(103, "Gaurav");
9. System.out.println("Initial list of elements: "+map);
10. //key-based removal
11. map.remove(100);
12. System.out.println("Updated list of elements: "+map);
13. //value-based removal
14. map.remove(101);
15. System.out.println("Updated list of elements: "+map);
16. //key-value pair based removal
17. map.remove(102, "Rahul");
18. System.out.println("Updated list of elements: "+map);
19. }
20. }

Output:

Initial list of elements: {100=Amit, 101=Vijay, 102=Rahul, 103=Gaurav}

Updated list of elements: {101=Vijay, 102=Rahul, 103=Gaurav}

Updated list of elements: {102=Rahul, 103=Gaurav}

Updated list of elements: {103=Gaurav}

### **Java HashMap example to replace() elements**

Here, we see different ways to replace elements.

1. **import** java.util.\*;
2. **class** HashMap3{
3. **public** **static** **void** main(String args[]){
4. HashMap<Integer,String> hm=**new** HashMap<Integer,String>();
5. hm.put(100,"Amit");
6. hm.put(101,"Vijay");
7. hm.put(102,"Rahul");
8. System.out.println("Initial list of elements:");
9. **for**(Map.Entry m:hm.entrySet())
10. {
11. System.out.println(m.getKey()+" "+m.getValue());
12. }
13. System.out.println("Updated list of elements:");
14. hm.replace(102, "Gaurav");
15. **for**(Map.Entry m:hm.entrySet())
16. {
17. System.out.println(m.getKey()+" "+m.getValue());
18. }
19. System.out.println("Updated list of elements:");
20. hm.replace(101, "Vijay", "Ravi");
21. **for**(Map.Entry m:hm.entrySet())
22. {
23. System.out.println(m.getKey()+" "+m.getValue());
24. }
25. System.out.println("Updated list of elements:");
26. hm.replaceAll((k,v) -> "Ajay");
27. **for**(Map.Entry m:hm.entrySet())
28. {
29. System.out.println(m.getKey()+" "+m.getValue());
30. }
31. }
32. }

Initial list of elements:

100 Amit

101 Vijay

102 Rahul

Updated list of elements:

100 Amit

101 Vijay

102 Gaurav

Updated list of elements:

100 Amit

101 Ravi

102 Gaurav

Updated list of elements:

100 Ajay

101 Ajay

102 Ajay

### **Difference between HashSet and HashMap**

HashSet contains only values whereas HashMap contains an entry(key and value).

### **Java HashMap Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** MapExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating map of Books
17. Map<Integer,Book> map=**new** HashMap<Integer,Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to map
23. map.put(1,b1);
24. map.put(2,b2);
25. map.put(3,b3);
27. //Traversing map
28. **for**(Map.Entry<Integer, Book> entry:map.entrySet()){
29. **int** key=entry.getKey();
30. Book b=entry.getValue();
31. System.out.println(key+" Details:");
32. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33. }
34. }
35. }

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

Output:

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

2 Details:

102 Data Communications and Networking Forouzan Mc Graw Hill 4

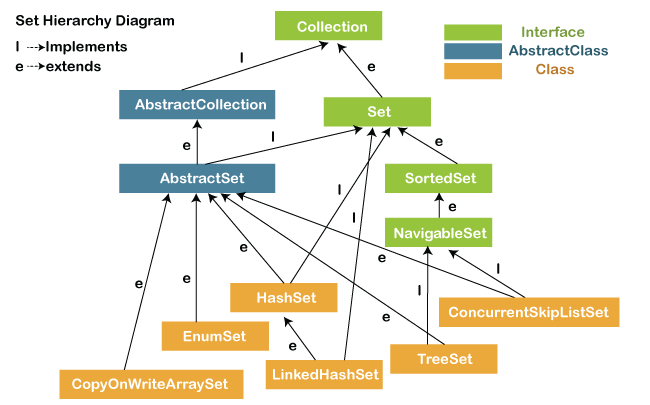
3 Details:

103 Operating System Galvin Wiley 6

**7. How to implement Set?**

**A:**

The **set** is an interface available in the **java.util** package. 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 is used to create the mathematical set. The set interface use collection interface's methods to avoid the insertion of the same elements. **SortedSet** and **NavigableSet** are two interfaces that extend the set implementation.

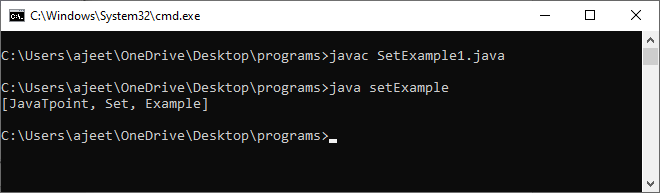


In the above diagram, the **NavigableSet** and **SortedSet** are both the interfaces. The **NavigableSet** extends the SortedSet, so it will not retain the insertion order and store the data in a sorted way.

### **SetExample1.java**

1. **import** java.util.\*;
2. **public** **class** setExample{
3. **public** **static** **void** main(String[] args)
4. {
5. // creating LinkedHashSet using the Set
6. Set<String> data = **new** LinkedHashSet<String>();
8. data.add("JavaTpoint");
9. data.add("Set");
10. data.add("Example");
11. data.add("Set");
13. System.out.println(data);
14. }
15. }

**Output:**



#### **Note: Throughout the section, we have compiled the program with file name and run the program with class name. Because the file name and the class name are different.**

## Operations on the Set Interface

On the Set, we can perform all the basic mathematical operations like intersection, union and difference.

Suppose, we have two sets, i.e., set1 = [22, 45, 33, 66, 55, 34, 77] and set2 = [33, 2, 83, 45, 3, 12, 55]. We can perform the following operation on the Set:

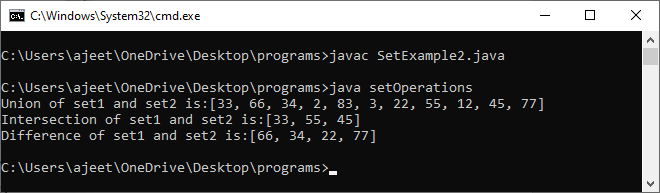
* **Intersection:** The intersection operation returns all those elements which are present in both the set. The intersection of set1 and set2 will be [33, 45, 55].
* **Union:** The union operation returns all the elements of set1 and set2 in a single set, and that set can either be set1 or set2. The union of set1 and set2 will be [2, 3, 12, 22, 33, 34, 45, 55, 66, 77, 83].
* **Difference:** The difference operation deletes the values from the set which are present in another set. The difference of the set1 and set2 will be [66, 34, 22, 77].

In set, **addAll()** method is used to perform the union, **retainAll()** method is used to perform the intersection and **removeAll()** method is used to perform difference. Let's take an example to understand how these methods are used to perform the intersection, union, and difference operations.

### **SetExample2.java**

1. **import** java.util.\*;
2. **public** **class** SetOperations
3. {
4. **public** **static** **void** main(String args[])
5. {
6. Integer[] A = {22, 45,33, 66, 55, 34, 77};
7. Integer[] B = {33, 2, 83, 45, 3, 12, 55};
8. Set<Integer> set1 = **new** HashSet<Integer>();
9. set1.addAll(Arrays.asList(A));
10. Set<Integer> set2 = **new** HashSet<Integer>();
11. set2.addAll(Arrays.asList(B));
13. // Finding Union of set1 and set2
14. Set<Integer> union\_data = **new** HashSet<Integer>(set1);
15. union\_data.addAll(set2);
16. System.out.print("Union of set1 and set2 is:");
17. System.out.println(union\_data);
19. // Finding Intersection of set1 and set2
20. Set<Integer> intersection\_data = **new** HashSet<Integer>(set1);
21. intersection\_data.retainAll(set2);
22. System.out.print("Intersection of set1 and set2 is:");
23. System.out.println(intersection\_data);
25. // Finding Difference of set1 and set2
26. Set<Integer> difference\_data = **new** HashSet<Integer>(set1);
27. difference\_data.removeAll(set2);
28. System.out.print("Difference of set1 and set2 is:");
29. System.out.println(difference\_data);
30. }
31. }

**Output:**



### **Description:**

In the above code, first, we create two arrays, i.e., A and B of type integer. After that, we create two set, i.e., set1 and set2 of type integer. We convert both the array into a list and add the elements of array A into set1 and elements of array B into set2.

For performing the union, we create a new set **union\_data** with the same element of the set1. We then call the addAll() method of set and pass the set2 as an argument to it. This method will add all those elements to the **union\_data** which are not present in it and gives the union of both sets.

For performing the intersection, we create a new set **intersection\_data** with the same element of the set1. We then call the retainAll() method of set and pass the set2 as an argument to it. This method will get all those elements from the **intersection\_data** which are present in set2 and store it in the intersection\_data. Now, the intersection\_data contains the intersect value of both the sets.

For performing the difference, we create a new set **difference\_data** with the same element of the set1. We then call the removeAll() method of set and pass the set2 as an argument to it. This method will remove all those elements from the **difference\_data** which are present in the set2 and gives the difference of both the sets.

## Set Methods

There are several methods available in the set interface which we can use to perform a certain operation on our sets. These methods are as follows:

### **1) add()**

The add() method insert a new value to the set. The method returns true and false depending on the presence of the insertion element. It returns false if the element is already present in the set and returns true if it is not present in the set.

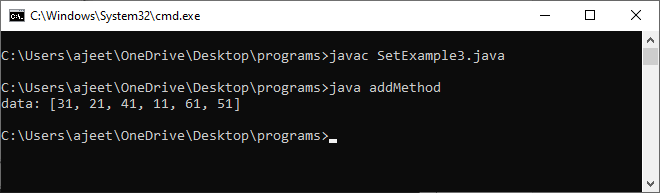
**Syntax:**

1. **boolean** add(type element).

**SetExample3.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **public** **class** addMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(11);
11. data.add(61);
12. data.add(51);
13. System.out.println("data: " + data);
14. }
15. }

**Output:**



### **2) addAll()**

The addAll() method appends all the elements of the specified collection to the set.

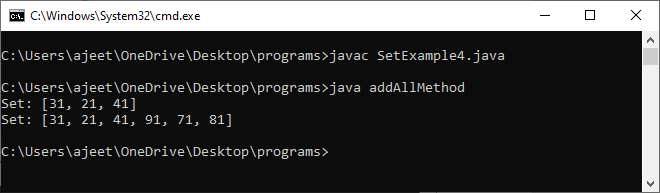
**Syntax:**

1. **boolean** addAll(Collection data)

**SetExample4.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** addAllMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. System.out.println("Set: " + data);
11. ArrayList<Integer> newData = **new** ArrayList<Integer>();
12. newData.add(91);
13. newData.add(71);
14. newData.add(81);
15. data.addAll(newData);
16. System.out.println("Set: " + data);
17. }
18. }

**Output:**



### **3) clear()**

The method removes all the elements from the set. It doesn't delete the reference of the set. It only deletes the elements of the set.

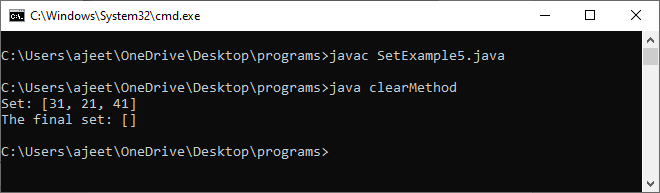
**Syntax:**

1. **void** clear()

**SetExample5.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **public** **class** clearMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
8. data.add(31);
9. data.add(21);
10. data.add(41);
11. System.out.println("Set: " + data);
13. data.clear();
14. System.out.println("The final set: " + data);
15. }
16. }

**Output:**



### **4) contains()**

The contains() method is used to know the presence of an element in the set. Its return value is true or false depending on the presence of the element.

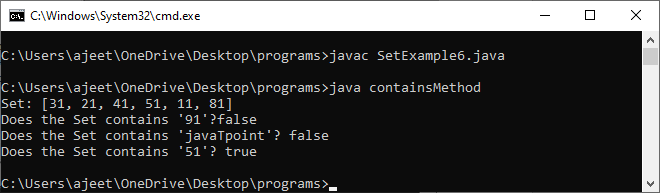
**Syntax:**

1. **boolean** contains(Object element)

**SetExample6.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** containsMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(51);
11. data.add(11);
12. data.add(81);
13. System.out.println("Set: " + data);
14. System.out.println("Does the Set contains '91'?" + data.contains(91));
15. System.out.println("Does the Set contains 'javaTpoint'? " + data.contains("4"));
16. System.out.println("Does the Set contains '51'? " + data.contains(51));
17. }
18. }

**Output:**



### **5) containsAll()**

The method is used to check whether all the elements of the collection are available in the existing set or not. It returns true if all the elements of the collection are present in the set and returns false even if one of the elements is missing in the existing set.

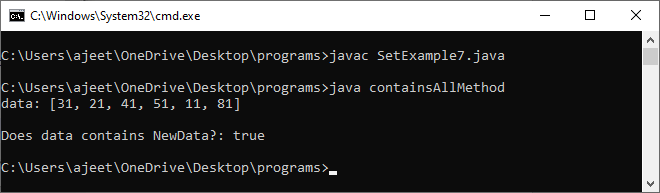
**Syntax:**

1. **public** **boolean** containsAll(Collection data)

**SetExample7.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** containsAllMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(51);
11. data.add(11);
12. data.add(81);
14. System.out.println("data: " + data);
16. Set<Integer> newData = **new** LinkedHashSet<Integer>();
17. newData.add(31);
18. newData.add(21);
19. newData.add(41);
21. System.out.println("\nDoes data contains newData?: "+ data.containsAll(newData));
23. }
24. }

**Output:**



### **6) hashCode()**

The method is used to derive the hash code value for the current instance of the set. It returns hash code value of integer type.

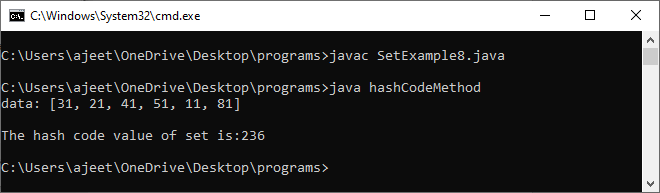
**Syntax:**

1. **public** **int** hashCode()

**SetExample8.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** hashCodeMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(51);
11. data.add(11);
12. data.add(81);
13. System.out.println("data: " + data);
14. System.out.println("\nThe hash code value of set is:"+ data.hashCode());
15. }
16. }

**Output:**



### **7) isEmpty()**

The isEmpty() method is used to identify the emptiness of the set . It returns true if the set is empty and returns false if the set is not empty.

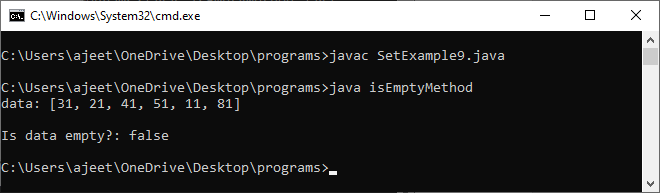
**Syntax:**

1. **boolean** isEmpty()

**SetExample9.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** isEmptyMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(51);
11. data.add(11);
12. data.add(81);
13. System.out.println("data: " + data);
14. System.out.println("\nIs data empty?: "+ data.isEmpty());
15. }
16. }

**Output:**



### **8) iterator()**

The iterator() method is used to find the iterator of the set. The iterator is used to get the element one by one.

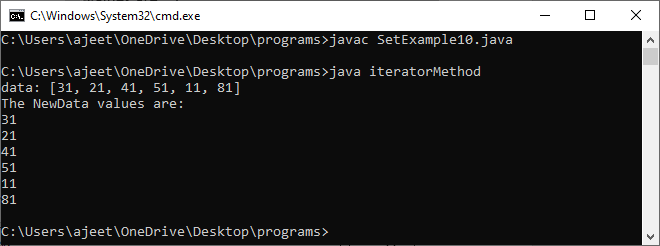
**Syntax:**

1. Iterator iterate\_value = set1.iterator();

**SetExample10.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** iteratorMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(51);
11. data.add(11);
12. data.add(81);
13. System.out.println("data: " + data);
15. Iterator newData = data.iterator();
16. System.out.println("The NewData values are: ");
17. **while** (newData.hasNext()) {
18. System.out.println(newData.next());
19. }
20. }
21. }

**Output:**



### **9) remove()**

The method is used to remove a specified element from the Set. Its return value depends on the availability of the element. It returns true if the element is available in the set and returns false if it is unavailable in the set.

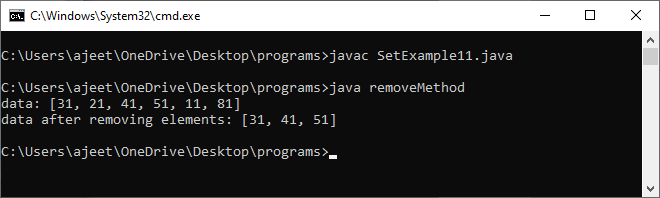
**Syntax:**

1. **boolean** remove(Object O)

**SetExample11.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** removeMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(51);
11. data.add(11);
12. data.add(81);
13. System.out.println("data: " + data);
15. data.remove(81);
16. data.remove(21);
17. data.remove(11);
18. System.out.println("data after removing elements: " + data);
19. }
20. }

**Output:**



### **11) removeAll()**

The method removes all the elements of the existing set from the specified collection.

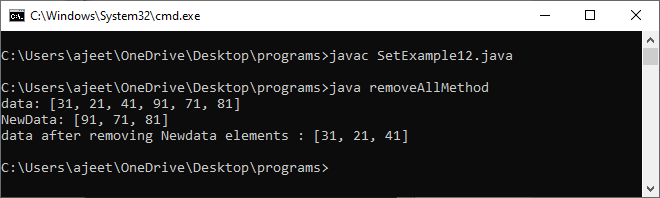
**Syntax:**

1. **public** **boolean** removeAll(Collection data)

**SetExample12.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** removeAllMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(91);
11. data.add(71);
12. data.add(81);
13. System.out.println("data: " + data);
15. ArrayList<Integer> newData = **new** ArrayList<Integer>();
16. newData.add(91);
17. newData.add(71);
18. newData.add(81);
19. System.out.println("NewData: " + newData);
21. data.removeAll(newData);
22. System.out.println("data after removing Newdata elements : " + data);
23. }
24. }

**Output:**



### **11) retainAll()**

The method retains all the elements from the set specified in the given collection.

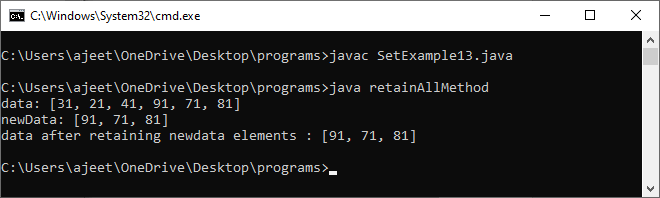
**Syntax:**

1. **public** **boolean** retainAll(Collection data)

**SetExample13.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** retainAllMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(91);
11. data.add(71);
12. data.add(81);
13. System.out.println("data: " + data);
15. ArrayList<Integer> newData = **new** ArrayList<Integer>();
16. newData.add(91);
17. newData.add(71);
18. newData.add(81);
19. System.out.println("newData: " + newData);
21. data.retainAll(newData);
22. System.out.println("data after retaining newdata elements : " + data);
23. }
24. }

**Output:**



### **12) size()**

The method returns the size of the set.

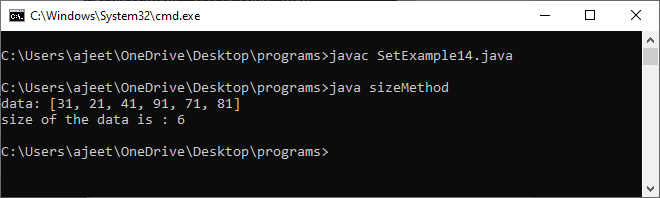
**Syntax:**

1. **int** size()

**SetExample14.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** sizeMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(91);
11. data.add(71);
12. data.add(81);
13. System.out.println("data: " + data);
15. System.out.println("size of the data is : " + data.size());
16. }
17. }

**Output:**



### **13) removeAll()**

The method is used to create an array with the same elements of the set.

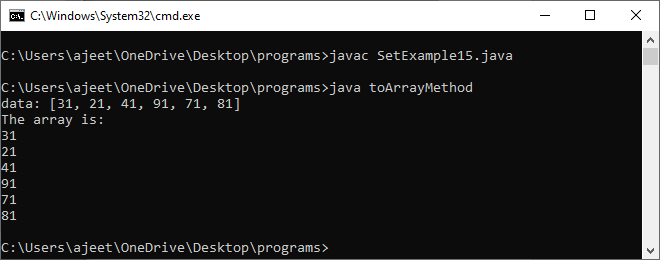
**Syntax:**

1. Object[] toArray()

**SetExample15.java**

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** toArrayMethod {
4. **public** **static** **void** main(String args[])
5. {
6. Set<Integer> data = **new** LinkedHashSet<Integer>();
7. data.add(31);
8. data.add(21);
9. data.add(41);
10. data.add(91);
11. data.add(71);
12. data.add(81);
13. System.out.println("data: " + data);
15. Object[] array\_data = data.toArray();
16. System.out.println("The array is:");
17. **for** (**int** i = 0; i < array\_data.length; i++)
18. System.out.println(array\_data[i]);
19. }
20. }

**Output:**



**8.MethodOver-riding with example?**

**A:**

# **Method Overriding in Java**

1. [Understanding the problem without method overriding](https://www.javatpoint.com/method-overriding-in-java#moverproblem)
2. [Can we override the static method](https://www.javatpoint.com/method-overriding-in-java#movercanstatic)
3. [Method overloading vs. method overriding](https://www.javatpoint.com/method-overriding-in-java#moverdiff)

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

### **Usage of Java Method Overriding**

* Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
* Method overriding is used for runtime polymorphism

#### **Rules for Java Method Overriding**

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).



### **Understanding the problem without method overriding**

Let's understand the problem that we may face in the program if we don't use method overriding.

1. //Java Program to demonstrate why we need method overriding
2. //Here, we are calling the method of parent class with child
3. //class object.
4. //Creating a parent class
5. **class** Vehicle{
6. **void** run(){System.out.println("Vehicle is running");}
7. }
8. //Creating a child class
9. **class** Bike **extends** Vehicle{
10. **public** **static** **void** main(String args[]){
11. //creating an instance of child class
12. Bike obj = **new** Bike();
13. //calling the method with child class instance
14. obj.run();
15. }
16. }

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

Output:

Vehicle is running

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

### **Example of method overriding**

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

1. //Java Program to illustrate the use of Java Method Overriding
2. //Creating a parent class.
3. **class** Vehicle{
4. //defining a method
5. **void** run(){System.out.println("Vehicle is running");}
6. }
7. //Creating a child class
8. **class** Bike2 **extends** Vehicle{
9. //defining the same method as in the parent class
10. **void** run(){System.out.println("Bike is running safely");}
12. **public** **static** **void** main(String args[]){
13. Bike2 obj = **new** Bike2();//creating object
14. obj.run();//calling method
15. }
16. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Bike2)

Output:

Bike is running safely

### **A real example of Java Method Overriding**

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.



#### **Java method overriding is mostly used in Runtime Polymorphism which we will learn in next pages.**

1. //Java Program to demonstrate the real scenario of Java Method Overriding
2. //where three classes are overriding the method of a parent class.
3. //Creating a parent class.
4. **class** Bank{
5. **int** getRateOfInterest(){**return** 0;}
6. }
7. //Creating child classes.
8. **class** SBI **extends** Bank{
9. **int** getRateOfInterest(){**return** 8;}
10. }
12. **class** ICICI **extends** Bank{
13. **int** getRateOfInterest(){**return** 7;}
14. }
15. **class** AXIS **extends** Bank{
16. **int** getRateOfInterest(){**return** 9;}
17. }
18. //Test class to create objects and call the methods
19. **class** Test2{
20. **public** **static** **void** main(String args[]){
21. SBI s=**new** SBI();
22. ICICI i=**new** ICICI();
23. AXIS a=**new** AXIS();
24. System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
25. System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
26. System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
27. }
28. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Test2)

Output:

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

### **Can we override static method?**

No, a static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

### **Why can we not override static method?**

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

### **Can we override java main method?**

No, because the main is a static method.

**9.How to implement List?**

**A:**

**List** in Java provides the facility to maintain the ordered collection. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.

The List interface is found in the java.util package and inherits the Collection interface. It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions. The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector. The ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

### **List Interface declaration**

1. **public** **interface** List<E> **extends** Collection<E>

### **Java List Methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index, E element) | It is used to insert the specified element at the specified position in a list. |
| boolean add(E e) | It is used to append the specified element at the end of a list. |
| boolean addAll(Collection<? extends E> c) | It is used to append all of the elements in the specified collection to the end of a list. |
| boolean addAll(int index, Collection<? extends E> c) | It is used to append all the elements in the specified collection, starting at the specified position of the list. |
| void clear() | It is used to remove all of the elements from this list. |
|  |  |
| boolean equals(Object o) | It is used to compare the specified object with the elements of a list. |
| int hashcode() | It is used to return the hash code value for a list. |
| E get(int index) | It is used to fetch the element from the particular position of the list. |
| boolean isEmpty() | It returns true if the list is empty, otherwise false. |
| int lastIndexOf(Object o) | It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element. |
| Object[] toArray() | It is used to return an array containing all of the elements in this list in the correct order. |
| <T> T[] toArray(T[] a) | It is used to return an array containing all of the elements in this list in the correct order. |
| boolean contains(Object o) | It returns true if the list contains the specified element |
| boolean containsAll(Collection<?> c) | It returns true if the list contains all the specified element |
| int indexOf(Object o) | It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element. |
| E remove(int index) | It is used to remove the element present at the specified position in the list. |  |
| boolean remove(Object o) | It is used to remove the first occurrence of the specified element. |  |
| boolean removeAll(Collection<?> c) | It is used to remove all the elements from the list. |  |
| void replaceAll(UnaryOperator<E> operator) | It is used to replace all the elements from the list with the specified element. |  |
| void retainAll(Collection<?> c) | It is used to retain all the elements in the list that are present in the specified collection. |  |
| E set(int index, E element) | It is used to replace the specified element in the list, present at the specified position. |  |
| void sort(Comparator<? super E> c) | It is used to sort the elements of the list on the basis of specified comparator. |  |
| Spliterator<E> spliterator() | It is used to create spliterator over the elements in a list. |  |
| List<E> subList(int fromIndex, int toIndex) | It is used to fetch all the elements lies within the given range. |  |
| int size() | It is used to return the number of elements present in the list. |  |

### **Java List vs ArrayList**

List is an interface whereas ArrayList is the implementation class of List.

### **How to create List**

The ArrayList and LinkedList classes provide the implementation of List interface. Let's see the examples to create the List:

1. //Creating a List of type String using ArrayList
2. List<String> list=**new** ArrayList<String>();
4. //Creating a List of type Integer using ArrayList
5. List<Integer> list=**new** ArrayList<Integer>();
7. //Creating a List of type Book using ArrayList
8. List<Book> list=**new** ArrayList<Book>();
10. //Creating a List of type String using LinkedList
11. List<String> list=**new** LinkedList<String>();

In short, you can create the List of any type. The ArrayList<T> and LinkedList<T> classes are used to specify the type. Here, T denotes the type.

### **Java List Example**

Let's see a simple example of List where we are using the ArrayList class as the implementation.

1. **import** java.util.\*;
2. **public** **class** ListExample1{
3. **public** **static** **void** main(String args[]){
4. //Creating a List
5. List<String> list=**new** ArrayList<String>();
6. //Adding elements in the List
7. list.add("Mango");
8. list.add("Apple");
9. list.add("Banana");
10. list.add("Grapes");
11. //Iterating the List element using for-each loop
12. **for**(String fruit:list)
13. System.out.println(fruit);
15. }
16. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=ListExample1)

Output:

Mango

Apple

Banana

Grapes

### **How to convert Array to List**

We can convert the Array to List by traversing the array and adding the element in list one by one using list.add() method. Let's see a simple example to convert array elements into List.

1. **import** java.util.\*;
2. **public** **class** ArrayToListExample{
3. **public** **static** **void** main(String args[]){
4. //Creating Array
5. String[] array={"Java","Python","PHP","C++"};
6. System.out.println("Printing Array: "+Arrays.toString(array));
7. //Converting Array to List
8. List<String> list=**new** ArrayList<String>();
9. **for**(String lang:array){
10. list.add(lang);
11. }
12. System.out.println("Printing List: "+list);
14. }
15. }

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

Output:

Printing Array: [Java, Python, PHP, C++]

Printing List: [Java, Python, PHP, C++]

### **How to convert List to Array**

We can convert the List to Array by calling the list.toArray() method. Let's see a simple example to convert list elements into array.

1. **import** java.util.\*;
2. **public** **class** ListToArrayExample{
3. **public** **static** **void** main(String args[]){
4. List<String> fruitList = **new** ArrayList<>();
5. fruitList.add("Mango");
6. fruitList.add("Banana");
7. fruitList.add("Apple");
8. fruitList.add("Strawberry");
9. //Converting ArrayList to Array
10. String[] array = fruitList.toArray(**new** String[fruitList.size()]);
11. System.out.println("Printing Array: "+Arrays.toString(array));
12. System.out.println("Printing List: "+fruitList);
13. }
14. }

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

Output:

Printing Array: [Mango, Banana, Apple, Strawberry]

Printing List: [Mango, Banana, Apple, Strawberry]

### **Get and Set Element in List**

The get() method returns the element at the given index, whereas the set() method changes or replaces the element.

1. **import** java.util.\*;
2. **public** **class** ListExample2{
3. **public** **static** **void** main(String args[]){
4. //Creating a List
5. List<String> list=**new** ArrayList<String>();
6. //Adding elements in the List
7. list.add("Mango");
8. list.add("Apple");
9. list.add("Banana");
10. list.add("Grapes");
11. //accessing the element
12. System.out.println("Returning element: "+list.get(1));//it will return the 2nd element, because index starts from 0
13. //changing the element
14. list.set(1,"Dates");
15. //Iterating the List element using for-each loop
16. **for**(String fruit:list)
17. System.out.println(fruit);
19. }
20. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=ListExample2)

**Output:**

Returning element: Apple

Mango

Dates

Banana

Grapes

### **How to Sort List**

There are various ways to sort the List, here we are going to use Collections.sort() method to sort the list element. The java.util package provides a utility class **Collections** which has the static method sort(). Using the **Collections.sort()** method, we can easily sort any List.

1. **import** java.util.\*;
2. **class** SortArrayList{
3. **public** **static** **void** main(String args[]){
4. //Creating a list of fruits
5. List<String> list1=**new** ArrayList<String>();
6. list1.add("Mango");
7. list1.add("Apple");
8. list1.add("Banana");
9. list1.add("Grapes");
10. //Sorting the list
11. Collections.sort(list1);
12. //Traversing list through the for-each loop
13. **for**(String fruit:list1)
14. System.out.println(fruit);
16. System.out.println("Sorting numbers...");
17. //Creating a list of numbers
18. List<Integer> list2=**new** ArrayList<Integer>();
19. list2.add(21);
20. list2.add(11);
21. list2.add(51);
22. list2.add(1);
23. //Sorting the list
24. Collections.sort(list2);
25. //Traversing list through the for-each loop
26. **for**(Integer number:list2)
27. System.out.println(number);
28. }
30. }

**Output:**

Apple

Banana

Grapes

Mango

Sorting numbers...

1

11

21

51

## Java ListIterator Interface

ListIterator Interface is used to traverse the element in a backward and forward direction.

### **ListIterator Interface declaration**

1. **public** **interface** ListIterator<E> **extends** Iterator<E>

### **Methods of Java ListIterator Interface:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(E e) | This method inserts the specified element into the list. |
| boolean hasNext() | This method returns true if the list iterator has more elements while traversing the list in the forward direction. |
| E next() | This method returns the next element in the list and advances the cursor position. |
| int nextIndex() | This method returns the index of the element that would be returned by a subsequent call to next() |
| boolean hasPrevious() | This method returns true if this list iterator has more elements while traversing the list in the reverse direction. |
| E previous() | This method returns the previous element in the list and moves the cursor position backward. |
| E previousIndex() | This method returns the index of the element that would be returned by a subsequent call to previous(). |
| void remove() | This method removes the last element from the list that was returned by next() or previous() methods |
| void set(E e) | This method replaces the last element returned by next() or previous() methods with the specified element. |

### **Example of ListIterator Interface**

1. **import** java.util.\*;
2. **public** **class** ListIteratorExample1{
3. **public** **static** **void** main(String args[]){
4. List<String> al=**new** ArrayList<String>();
5. al.add("Amit");
6. al.add("Vijay");
7. al.add("Kumar");
8. al.add(1,"Sachin");
9. ListIterator<String> itr=al.listIterator();
10. System.out.println("Traversing elements in forward direction");
11. **while**(itr.hasNext()){
13. System.out.println("index:"+itr.nextIndex()+" value:"+itr.next());
14. }
15. System.out.println("Traversing elements in backward direction");
16. **while**(itr.hasPrevious()){
18. System.out.println("index:"+itr.previousIndex()+" value:"+itr.previous());
19. }
20. }
21. }

Output:

Traversing elements in forward direction

index:0 value:Amit

index:1 value:Sachin

index:2 value:Vijay

index:3 value:Kumar

Traversing elements in backward direction

index:3 value:Kumar

index:2 value:Vijay

index:1 value:Sachin

index:0 value:Amit

### **Example of List: Book**

Let's see an example of List where we are adding the Books.

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** ListExample5 {
15. **public** **static** **void** main(String[] args) {
16. //Creating list of Books
17. List<Book> list=**new** ArrayList<Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications and Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to list
23. list.add(b1);
24. list.add(b2);
25. list.add(b3);
26. //Traversing list
27. **for**(Book b:list){
28. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29. }
30. }
31. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=ListExample5)

Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications and Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

**10.Interface with Example?**

**A:**

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways −

* An interface can contain any number of methods.
* An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
* The byte code of an interface appears in a **.class** file.
* Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

* You cannot instantiate an interface.
* An interface does not contain any constructors.
* All of the methods in an interface are abstract.
* An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
* An interface is not extended by a class; it is implemented by a class.
* An interface can extend multiple interfaces.

## Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface −

### **Example**

Following is an example of an interface −

/\* File name : NameOfInterface.java \*/

import java.lang.\*;

// Any number of import statements

public interface NameOfInterface {

// Any number of final, static fields

// Any number of abstract method declarations\

}

Interfaces have the following properties −

* An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.
* Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
* Methods in an interface are implicitly public.

### **Example**

/\* File name : Animal.java \*/

interface Animal {

public void eat();

public void travel();

}

## Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

### **Example**

/\* File name : MammalInt.java \*/

public class MammalInt implements Animal {

public void eat() {

System.out.println("Mammal eats");

}

public void travel() {

System.out.println("Mammal travels");

}

public int noOfLegs() {

return 0;

}

public static void main(String args[]) {

MammalInt m = new MammalInt();

m.eat();

m.travel();

}

}

This will produce the following result −

### **Output**

Mammal eats

Mammal travels

When overriding methods defined in interfaces, there are several rules to be followed −

* Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
* The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
* An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementation interfaces, there are several rules −

* A class can implement more than one interface at a time.
* A class can extend only one class, but implement many interfaces.
* An interface can extend another interface, in a similar way as a class can extend another class.

## Extending Interfaces

An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces.

### **Example**

// Filename: Sports.java

public interface Sports {

public void setHomeTeam(String name);

public void setVisitingTeam(String name);

}

// Filename: Football.java

public interface Football extends Sports {

public void homeTeamScored(int points);

public void visitingTeamScored(int points);

public void endOfQuarter(int quarter);

}

// Filename: Hockey.java

public interface Hockey extends Sports {

public void homeGoalScored();

public void visitingGoalScored();

public void endOfPeriod(int period);

public void overtimePeriod(int ot);

}

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

## Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as −

### **Example**

public interface Hockey extends Sports, Event

## Tagging Interfaces

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as −

### **Example**

package java.util;

public interface EventListener

{}

An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces −

**Creates a common parent** − As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

**Adds a data type to a class** − This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

**11.waits in selenium:**

**A:**

You might have come across wait commands while writing your first Selenium program. In this article, you would be learning about what exactly Selenium Waits is. You would be covering various types and other necessary factors one needs to understand to get started with Selenium Waits.

## What is Selenium Waits?

Waits in Selenium is one of the important pieces of code that executes a **test case.** It runs on certain commands called scripts that make a page load through it. Selenium Waits makes the pages **less vigorous** and **reliable.** It provides various types of wait options adequate and suitable under favorable conditions. This ensures you don't mess up and get ended into failed scripts while performing automation testing with it.

Elaborately, [Selenium](https://www.javatpoint.com/selenium-tutorial) Waits helps the user to troubleshoot various issues while page redirection across different web pages. It is achieved by refreshing the entire web page and reloading it with new elements. At times, there's a call from **Ajax** as well. Thus, some time lag might exist while reloading pages and reflecting elements present on the web pages after refreshing.

Another instance to understand Selenium Waits is navigating web pages back and forth with the **navigate()** command. This navigate() method comes from [**WebDriver,**](https://www.javatpoint.com/selenium-webdriver) whose main task is to simulate and manifest real-time scenarios like navigating between web pages concerning browsing history.

## Why Do You Need Waits In Selenium?

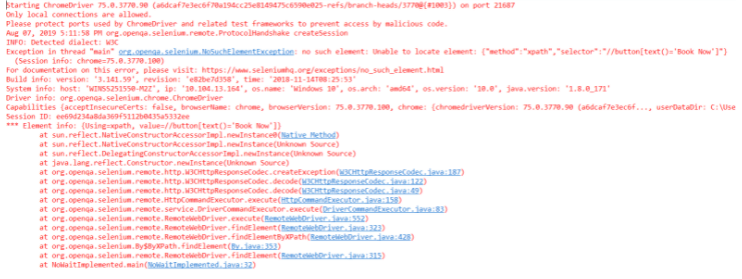
Today, most of the modern application's front-end is built on either [Ajax](https://www.javatpoint.com/ajax-tutorial) or [JavaScript](https://www.javatpoint.com/javascript-tutorial), followed by popular frameworks like Angular, React, or any other, which takes some time for loading elements on the web page. Hence, in such a case, Selenium throws an **'ElementNotVisibleException'** message when you tend to locate an element present in your script which is still not loaded on the web page.

To clarify, you can look at the below code snippet where automation testing with Selenium is executed.

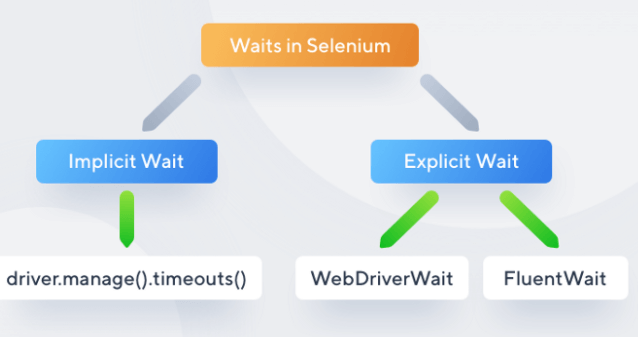
The given code will help you showcase the same problem as you execute automation testing with Selenium. An example of easemytrip.com is used, where the posting user selects the 'From' and 'To' destination with a date of journey. The web application takes a certain time to load the required flight details. In this case, without applying Wait, the user tends to book the first flight from the list. Since the page hasn't loaded yet, the script failed to find the 'book now button. It results in throwing a **'NoSuchElementException'.** It is given below:

1. **import** org.open.selenium.By;
2. **import** org.openqa.selenium.JavascriptExecutor;
3. **import** org.openqa.selenium.Keys;
4. **import** org.openqa.selenium.WebDriver;
5. **import** org.openqa.selenium.chrome.ChromeDriver;
6. **import**
7. com.gargoylesoftware.htmlunit.javascript.background.JavaScriptExec
8. utor;
9. **public** **class** NoWaitImplemented {
10. **public** **static** **void** main(String[] args) **throws** InterruptedException {
11. System.setProperty("webdriver.chrome.driver",
12. ".\\Driver\\chromedriver.exe");
13. WebDriver driver=**new** ChromeDriver();
14. driver.manage().window().maximize();
15. driver.get("https://www.easemytrip.com/");
16. driver.findElement(By.id("FromSector\_show")).sendKeys("Delhi",
17. Keys.ENTER);
18. driver.findElement(By.id("Editbox13\_show")).sendKeys("Mumbai",
19. Keys.ENTER);
20. driver.findElement(By.id("ddate")).click();
21. driver.findElement(By.id("snd\_4\_08/08/2019")).click();
22. driver.findElement(By.className("src\_btn")).click();
23. driver.findElement(By.xpath("//button[text()='Book Now']")).click();
24. }
25. }

The above code snippet depicts the same problem while executing automation testing with Selenium. In this code snippet, you can see an example of **"easemytrip.com,"** where the user will select **'From'** and **'To'** destination selection with a journey date. The web application takes a certain loading time to load the available flights based on the selected input fields provided by the user. In this case, the user might select and book only the first flight from the list. Since the page is still loading, the script has failed to find the 'Book Now' button. This directly throws a **'NoSuchElementExpection'** return status with the following output shown below.



## Types of Waits in Selenium



### **Implicit Waits**

The main function of implicit Wait is to tell the web driver to wait for some time before throwing a **"No Such Element Exception".** Its default setting is knocked at zero. Once the time is set, the driver automatically will wait for the amount of time defined by you before throwing the above-given exception.

**Syntax:**

1. driver.manage().timeouts().implicitlyWait(TimeOut,
2. TimeUnit.SECONDS);

To understand how implicit wait works, let's consider an example.

1. **package** JavaTpoint;
2. **import** java.util.concurrent.TimeUnit;
3. **import** org.openqa.selenium.By;
4. **import** org.openqa.selenium.WebDriver;
5. **import** org.openqa.selenium.WebElement;
6. **import** org.openqa.selenium.chrome.ChromeDriver;
7. **import** org.openqa.selenium.support.ui.ExpectedConditions;
8. **import** org.openqa.selenium.support.ui.WebDriverWait;
9. **public** **class** ImplicitWait{
10. **public** **static** **void** main(String[] args) **throws** InterruptedException
11. {
12. System.setProperty("webdriver.chrome.driver", "C:Selenium-java-
13. javaTpointchromedriver\_win32chromedriver.exe");
14. WebDriver driver = **new** ChromeDriver();
15. driver.manage().window().maximize();
16. driver.manage().deleteAllCookies();
17. driver.manage().timeouts().pageLoadTimeout(40,
18. TimeUnit.SECONDS); // pageload timeout
19. driver.manage().timeouts().implicitlyWait(20, TimeUnit.SECONDS);
20. // Implicit Wait for 20 seconds
21. driver.get("https://login.google.com/");
22. driver.findElement(By.xpath("//input[@id='login-
23. username']")).sendKeys("JavaTpoint.com"); //Finding element and
24. sending values
25. Thread.sleep(1000);
26. driver.findElement(By.xpath("//input[@id='login-signin']")).click(); //Clicking on the next button **if** element is located
27. }
28. }

In the code snippet given above, the Implicit Wait is defined for only **20 seconds,** implying that the output will load or arrive within the maximum waiting time of 20 seconds for the particular element.

#### **Note: Implicit Wait is globally applied. It is readily available for driver instance. It also means that if the driver is having interaction with One thousand elements in the meantime, the implicit Wait will be applicable only for those 1000 elements. It cannot go beyond that.**

### **Explicit Waits**

Explicit Waits also known as **Dynamic Waits** because it is highly specific conditioned. It is implemented by WebDriverWait class. To understand why you need Explicit Wait in Selenium, you must go through the basic knowledge of the wait statements in a program. In simple terms, you must know some conditions. Such conditions have been created to give you a gist of the Explicit Waits and why they are important.

**Condition 1:**

Suppose you have a web page consisting of a login form that takes input and loads the Home or Main page content. This page is dynamic because of the time constraints and network frequency, sometimes taking 10 seconds or maybe 15 seconds to load completely. Explicit Wait comes in handy in such cases and allows you to wait until the page is not present to display.

**Condition 2:**

Consider that you are working on an application that is travel themed and users fill the web form and submit it using submit button. Now, you might need to wait until and unless the specific data is not displayed. In such a case, Explicit Wait becomes helpful by waiting until a specific period for the set of elements that are not displayed yet.

**Syntax:**

1. WebDriverWait wait=**new**
2. ebDriverWait(WebDriveReference,TimeOut);

The above syntax justifies an object of WebDriver Wait and is passed to the driver's preference, and the timeout is taken as a parameter. To understand this more broadly, consider the below sample application.

1. **package** JavaTpoint;
2. **import** java.util.concurrent.TimeUnit;
3. **import** org.openqa.selenium.By;
4. **import** org.openqa.selenium.WebDriver;
5. **import** org.openqa.selenium.WebElement;
6. **import** org.openqa.selenium.chrome.ChromeDriver;
7. **import** org.openqa.selenium.support.ui.ExpectedConditions;
8. **import** org.openqa.selenium.support.ui.WebDriverWait;
9. **public** **class** Locators {
10. **public** **static** **void** main(String[] args) **throws** InterruptedException {
11. System.setProperty("webdriver.chrome.driver", "C:Selenium-java-
12. javatpointchromedriver\_win32chromedriver.exe");
13. WebDriver driver = **new** ChromeDriver();
14. driver.manage().window().maximize();
15. driver.manage().deleteAllCookies();
16. driver.manage().timeouts().pageLoadTimeout(40,
17. TimeUnit.SECONDS);
18. driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
19. driver.get("https://www.facebook.com/");
20. WebElement firstname= driver.findElement(By.name("firstname"));
21. WebElement lastname= driver.findElement(By.name("lastname"));
22. sendKeys(driver, firstname, 10, "Edureka");
23. sendKeys(driver, lastname, 20, "Edureka");
24. WebElement forgotAccount=
25. driver.findElement(By.linkText("Forgotten account?"));
26. clickOn(driver,forgotAccount, 10);
27. driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
28. }
29. **public** **static** **void** sendKeys(WebDriver driver1, WebElement element,
30. **int** timeout, String value){
31. **new** WebDriverWait(driver1,
32. timeout).until(ExpectedConditions.visibilityOf(element));
33. element.sendKeys(value);
34. }
35. **public** **static** **void** clickOn(WebDriver driver1, WebElement element,
36. **int** timeout)
37. {
38. **new** WebDriverWait(driver1,
39. timeout).until(ExpectedConditions.elementToBeClickable(element));
40. element.click();
41. }
42. }

In the sample snippet given above, you can see Facebook sign-up credentials using locators have been created. Further, a generic utility function is created to make all the elements available for the Explicit Wait. Also, sendKeys() method is defined for the particular text field that would internally provide explicit Wait. Inside the sendKeys() method, there are some expected conditions for the element. This means that the driver is being asked to wait for 20 seconds until and unless the expected condition of the element is visible. Furthermore, the main purpose of applying the sendKeys() method is to take the first name and the last name, and that will be passed to the driver. The timeout is defined to be 10 seconds for the first name and the last name.

When the above program is executed, the Chrome driver will launch Chrome, and it will navigate through facebook.com to take the mentioned values. It is, although not mandatory, to explicitly set the timeout for a particular value since it is changeable. Another advantage to adding up here is that once you define the timeout for 10 seconds, it becomes applicable for all the elements present on the web page, and then it cannot be modified. The same concept works with the onClick() method defined in the above program, but this method is constrained only to links. This is the best way to execute, understand and execute Explicit Wait.

#### **Note: Dynamic weights are considered dynamic because that if you take Implicit, Explicit, or Fluent waits, and define the timeout method to be 20 seconds, and if the element takes only 5 seconds, the remaining 15 seconds are not considered. It will not wait for the whole 20 seconds.**

## Difference between Implicit and Explicit

1. Implicit Wait applies to all the elements in the script, while Explicit Wait is applicable only for those values which are to be defined by the user.
2. Implicit Wait needs specifying "ExpectedConditions" on the located element, while Explicit Wait doesn't need to be specified with this condition.
3. Implicit Wait needs time frame specification in terms of methods like element visibility, clickable element, and the elements that are to be selected. In contrast, Explicit Wait is dynamic and needs no such specifications.

### **Fluent Wait**

Fluent Wait is quite similar to explicit Wait. It is similar in terms of management and functioning. In Fluent Wait, you can perform wait for action for an element only when you are unaware of the time it might take to be clickable or visible. Few differential factors that Fluent offers are as follows:

**The pooling frequency**

The pooling frequency in the case of Explicit is 500 milliseconds. But, using Fluent Wait, this pooling frequency can be changed to any value based upon your need. This usually means telling the script to keep an eye on the element after every 'x' seconds.

**Ignore Exception**

While pooling, if an element is not found, you can ignore some expectations like 'NoSuchElement'. Apart from this factor, similar to Explicit and Implicit Wait, you can define the amount of time for the element to be actionable or visible.

**Syntax:**

1. Wait**<WebDriver>** fluentWait = new FluentWait**<WebDriver>**(driver)
2. .withTimeout(60, SECONDS) // this defines the total amount of
3. time to wait for
4. .pollingEvery(2, SECONDS) // this defines the polling frequency
5. .ignoring(NoSuchElementException.class); // this defines the
6. exception to ignore
7. WebElement foo = fluentWait.until(new Function**<WebDriver**,
8. WebElement**>**()
9. {
10. public WebElement apply(WebDriver driver)  //in this method
11. defined your own subjected conditions for which we need to wait for
12. {  return driver.findElement(By.id("foo"));
13. }});

Although the syntax appears to be very complex, it comes in handy when you start using it. This is why probably all the software testers prefer using Explicit and Implicit Wait because of the simple syntax. Fluent waits appear complex because it doesn't provide predefined conditions that need to apply to the elements. Rather, Fluent Wait defines its condition within the apply method.

1. **package** javaTpoint.test;
2. **import** org.testng.annotations.Test;
3. **import** java.util.NoSuchElementException;
4. **import** java.util.concurrent.TimeUnit;
5. **import** java.util.function.Function;
6. **import** org.openqa.selenium.By;
7. **import** org.openqa.selenium.WebDriver;
8. **import** org.openqa.selenium.WebElement;
9. **import** org.openqa.selenium.chrome.ChromeDriver;
10. **import** org.openqa.selenium.support.ui.ExpectedConditions;
11. **import** org.openqa.selenium.support.ui.FluentWait;
12. **import** org.openqa.selenium.support.ui.Wait;
13. **import** org.openqa.selenium.support.ui.WebDriverWait;
14. **import** org.testng.annotations.Test;
15. **public** **class** AppTest3 {
16. **protected** WebDriver driver;
17. @Test
18. **public** **void** javaTpointTutorial() **throws** InterruptedException
19. {
20. System.setProperty ("webdriver.chrome.driver",".\\chromedriver.exe" );
21. String eTitle = "Demo javaTpoint Page";
22. String aTitle = "" ;
23. driver = **new** ChromeDriver();
24. // launch Chrome and redirect it to the Base URL
25. driver.get("https://javatpoint.com/home" );
26. //Maximizes the browser window
27. driver.manage().window().maximize() ;
28. //get the actual value of the title
29. aTitle = driver.getTitle();
30. //compare the actual title with the expected title
31. **if** (aTitle.contentEquals(eTitle))
32. {
33. System.out.println( "Test Passed") ;
34. }
35. **else** {
36. System.out.println( "Test Failed" );
37. }
38. Wait<WebDriver> wait = **new** FluentWait<WebDriver>(driver)
39. .withTimeout(30, TimeUnit.SECONDS)
40. .pollingEvery(5, TimeUnit.SECONDS)
41. .ignoring(NoSuchElementException.**class**);
42. WebElement clickseleniumlink = wait.until(**new**
43. Function<WebDriver, WebElement>(){
44. **public** WebElement apply(WebDriver driver ) {
45. **return**
46. driver.findElement(By.xpath("/html/body/div[1]/section/div[2]/div/di
47. v[1]/div/div[1]/div/div/div/div[2]/div[2]/div/div/div/div/div[1]/div/div/a/i"));
48. }
49. });
50. //click on the selenium link
51. clickseleniumlink.click();
52. //close~ browser
53. driver.close() ;
54. }
55. }

In the above code snippet, you can observe that the fluent Wait has been set for the timeout of 30 seconds, and the frequency is kept at 5 seconds by "NoSuchElementException". In another instance, a new function to identify the web element on the former page is created. If the element to found to fall in the same time frame, it may perform all the operations. If for some reason, any error shoots up, the function returns "ElementNotVisibleException".

## Cloud applications of Selenium Waits

Most testers prefer performing automation testing for the projects they are working on using some cloud-based service providers designed to suit Selenium. One such popular cloud-based testing method is LamdbaTest. LambdaTest is a powerful cross-platform browser testing tool.

Consider that you are running a cloud-based Selenium Grid like LambdaTest, and this is where the Selenium complex test suit comes up with a rapid timeout issue. You also have the facility of pausing the WebDriver for 90 seconds which can also be chosen as a default time limit, thereby avoiding timeout errors as the automation script runs successfully.

## Final Thoughts

Selenium Waits help the users to design scripts with reliability and fewer dependencies. You might choose whichever wait you want to proceed with by ensuring the business unit achieves the result and the purpose behind automation testing using Selenium. Another important aspect to note is ensuring that you are not keeping Selenium Waits unnecessarily in your application. Initially, there was the option of threads in the previous versions of Selenium which is now replaced with Waits. Selenium waits offer dynamic testing methods and an optimal environment for testing using platforms like JUnit or other testing mechanisms. In this article, you saw how the difference between the former waits like implicit, explicit, and fluent. You also learned why Fluent Wait is not preferred though highly sophisticated but syntactically tough for developers to carry out unit testing methods using Selenium. Later you learned through example code snippets how the aspects of Selenium Wait work in close integration with each other.

**12.Xpath-Axes:**

**A:**

As we know that path defines the location of a node using absolute or relative path. In the same manner XPath axes are used to identify elements by their relationship like parent, child, sibling, etc. Axes refer to axis on which elements are lying relative to an element.

A list of various Axis values:

|  |  |  |
| --- | --- | --- |
| **Index** | **Axis** | **Description** |
| 1) | ancestor | It specifies the ancestors of the current nodes which include the parents up to the root node. |
| 2) | ancestor-or-self | It specifies the current node and its ancestors. |
| 3) | attribute | It specifies the attributes of the current node. |
| 4) | child | It specifies the children of the current node. |
| 5) | descendant | It specifies the descendants of the current node i.e. the node's children up to the leaf node(no more children). |
| 6) | descendant-or-self | It specifies the current node and it's descendants. |
| 7) | following | It specifies all nodes that come after the current node. |
| 8) | following-sibling | It specifies the following siblings of the context node. Siblings are at the same level as the current node and share it's parent. |
| 9) | namespace | It specifies the namespace of the current node. |
| 10) | parent | It specifies the parent of the current node. |
| 11) | preceding | It specifies all nodes that come before the current node (i.e. before it's opening tag). |
| 12) | self | It specifies the current node. |

XPath Axes Example

1. **<p><xsl:value-of** select = "firstname"**/></p>**
2. **<xsl:value-of** select = "/class/student/preceding-sibling::comment()"**/>**

Here the firstname is related to employee node of the XML document named employee.xml

Let's create an XML document named "employee.xml" and its stylesheet document named "employee.xsl" which uses the XPath expression.

**Employee.xml**

1. **<?xml** version = "1.0"**?>**
2. **<?xml-stylesheet** type = "text/xsl" href = "employee.xsl"**?>**
3. **<class>**
4. **<employee** id = "001"**>**
5. **<firstname>**Abhiram**</firstname>**
6. **<lastname>**Kushwaha**</lastname>**
7. **<nickname>**Manoj**</nickname>**
8. **<salary>**15000**</salary>**
9. **</employee>**
10. **<employee** id = "002"**>**
11. **<firstname>**Akash**</firstname>**
12. **<lastname>**Singh**</lastname>**
13. **<nickname>**Bunty**</nickname>**
14. **<salary>**25000**</salary>**
15. **</employee>**
16. **<employee** id = "003"**>**
17. **<firstname>**Brijesh**</firstname>**
18. **<lastname>**Kaushik**</lastname>**
19. **<nickname>**Ballu**</nickname>**
20. **<salary>**20000**</salary>**
21. **</employee>**
22. **<employee** id = "004"**>**
23. **<firstname>**Zoya**</firstname>**
24. **<lastname>**Mansoori**</lastname>**
25. **<nickname>**Sonam**</nickname>**
26. **<salary>**30000**</salary>**
27. **</employee>**
28. **</class>**

**Employee.xsl**

1. **<?xml** version = "1.0" encoding = "UTF-8"**?>**
2. **<xsl:stylesheet** version = "1.0"**>**
3. xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"**>**
4. **<xsl:template** match = "/" **>**
5. **<html>**
6. **<body>**
7. **<xsl:value-of** select = "/class/employee/preceding-sibling::comment()"**/>**
8. **<br/>**
9. **<xsl:text>**First Employee: **</xsl:text>**
10. **<xsl:value-of** select = "/class/employee/child::firstname" **/>**
11. **</body>**
12. **</html>**
13. **</xsl:template>**
14. **</xsl:stylesheet>**

**Output:**

