Java

[Wrapper Classes 5](#_Toc132531189)

[Object Class 6](#_Toc132531190)

[Object class Example 7](#_Toc132531191)

[Strings In Java 10](#_Toc132531192)

[**String Class** 10](#_Toc132531193)

[**Methods of String.** 12](#_Toc132531194)

[**SCP: String constant pool** 13](#_Toc132531195)

[**Immutable String** 16](#_Toc132531196)

[**String comparison** 18](#_Toc132531197)

[**String buffer** 18](#_Toc132531198)

[Interface 22](#_Toc132531199)

[Comparator interface 23](#_Toc132531200)

[Collection 24](#_Toc132531201)

[**List interface** 25](#_Toc132531202)

[**ArrayList** 26](#_Toc132531203)

[**LinkedList** 14](#_Toc132531204)

[**ArrayList and LinkedList Difference** 19](#_Toc132531205)

[**Vector** 21](#_Toc132531206)

[**Stack** 23](#_Toc132531207)

[**Iterators in Java** 26](#_Toc132531208)

[**Enumeration in Java** 26](#_Toc132531209)

[**Iterator** 28](#_Toc132531210)

[**Set** 34](#_Toc132531211)

[**What is Set** 34](#_Toc132531212)

[**Set Methods** 35](#_Toc132531213)

[**HashSet** 35](#_Toc132531214)

[**HashSet Code Example** 37](#_Toc132531215)

[**How HashSet add method works** 40](#_Toc132531216)

[**LinkedHashSet** 42](#_Toc132531217)

[**TreeSet** 44](#_Toc132531218)

[**Code Example TreeSet** 44](#_Toc132531219)

[**Null inside TreeSet** 46](#_Toc132531220)

[Difference: HashSet, LinkedHashSet, TreeSet 47](#_Toc132531221)

[Difference: List and Set 47](#_Toc132531222)

[Java Map Interface 49](#_Toc132531223)

[Map Interface 49](#_Toc132531224)

[**What is Map** 49](#_Toc132531225)

[**Map Methods** 50](#_Toc132531226)

[**HashMap** 51](#_Toc132531227)

[**Concurrency** 58](#_Toc132531228)

[**Multithread** 58](#_Toc132531229)

[Example of concurrency 58](#_Toc132531230)

[**Life cycle of a thread** 58](#_Toc132531231)

[Three ways to create thread 59](#_Toc132531232)

[**By inheriting your class from the Thread class** 59](#_Toc132531233)

[**By implementing the Runnable interface in your class** 60](#_Toc132531234)

[**By using the method reference to a method that takes no parameters and returns void** 61](#_Toc132531235)

[**Thread synchronization** 63](#_Toc132531236)

[Producer consumer problem 70](#_Toc132531237)

[Solution to producer consumer problem. 78](#_Toc132531238)

[Fork-Join 80](#_Toc132531239)

[The Java volatile Visibility Guarantee 89](#_Toc132531240)

[Deadlock Solution Example 91](#_Toc132531241)

[**Java 8** 94](#_Toc132531242)

[Interface default and static methods. 94](#_Toc132531243)

[Why default method? 94](#_Toc132531244)

[Output: Compile Time Error : class Multitalented inherits unrelated defaults for write() from types Poet and Writer You can see that both Poet and Writer interface have a default method write() and when we have created a class Multitalented, which implements both Poet and Writer it gets the write() method from both super interface Poet and Writer. Now, the problem is not that we have implemented two interfaces with default methods of the same name. The Actual problems come when we created an object of a Multitalented class and called the write() method on its object 97](#_Toc132531245)

[How to avoid Diamond Problem With Default Methods in Java 8 97](#_Toc132531246)

[**Lambda Expression in Java 8** 98](#_Toc132531247)

[Functional Interfaces 98](#_Toc132531248)

[Structure of a Lambda Expression 99](#_Toc132531249)

[Example Lambda Expressions 99](#_Toc132531250)

[Lambda Expression – Key Notes 100](#_Toc132531251)

[1. Lambda Expression Hello World 100](#_Toc132531252)

[2. Lambda Expression with Runnable 102](#_Toc132531253)

[3. Accessing Local and Class Variables in Lambda Expression 105](#_Toc132531254)

[4. Redeclaring Local Variables in Lambda Expression 106](#_Toc132531255)

[5. Modify Local Variables in the Lambda Expression 106](#_Toc132531256)

[6. Lambdas and the 'this' keyword 107](#_Toc132531257)

[7. Lambdas and exceptions 108](#_Toc132531258)

[8) Predefined functional interfaces 109](#_Toc132531259)

[Functional Interfaces: Predicate, Consumer, Function, and Supplier 111](#_Toc132531260)

[**Java 8 method references, double colon (::) operator** 118](#_Toc132531261)

[1. Static method 119](#_Toc132531262)

[2. Reference to an instance method of a particular object 123](#_Toc132531263)

[3. Reference to an instance method of an arbitrary object of a particular type. 125](#_Toc132531264)

[BiFunction<T, U, R> 126](#_Toc132531265)

[Example : Reference to an instance method of an arbitrary object of a particular type. 127](#_Toc132531266)

[4. Reference to a constructor. 128](#_Toc132531267)

# Wrapper Classes

# Object Class

Object class defined in java.lang package is the superclass of all other classes defined in Java programming language. Every class extends from the Object class either directly or indirectly.

**Methods define in Object class.**

clone() - Creates and returns a copy of this object.

equals() - Indicates whether some other object is "equal to" this one.

finalize() - Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

getClass() - Returns the runtime class of an object.

hashCode() - Returns a hash code value for the object.

notify() - Wakes up a single thread that is waiting on this object's monitor.

notifyAll() - Wakes up all threads that are waiting on this object's monitor.

toString() - Returns a string representation of the object.

wait() - Causes current thread to wait until another thread invokes the notify() or notifyAll() method for this object.

***equals()*** and ***hashcode()***.

*equals()* and *hashcode() are one of the two method of cosmic class java.lang.Object*

*Used to compare the objects.*

***equals()*** :

Signature : public boolean equals() { }

The default implementation provided by the JDK is based on memory location — two objects are equal if and only if they are stored or referred in the same memory address (override equals() method to check the equality as per your requirement.)

***hashcode () :***

Signature : public boolean equals() { }

*T*his method returns a random integer that is unique for each instance. This integer might change between several executions of the application and won't stay the same.

# Object class Example

**Example 1**

**package** com.corejava.collections.learning;

**public** **class** ObjectTest {

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

Student1 obj1=**new** Student1(1,"Ramesh");

Student1 obj2=**new** Student1(2,"Suresh");

Student1 obj3=**new** Student1(1,"Ramesh");

/\*

\* If we don't override equals() method in Student1 class then the default equals method from the Object class check the reference of the object e.g return (obj1 == obj2);

Try commenting equals method. If do so then the below equality will return false as both bj1 and obj2 have different reference (as they are created by new)

\*/

System.***out***.println("obj1.equals(obj3) : "+obj1.equals(obj3));

System.***out***.println("Obj1 hashCode:"+obj1.hashCode());

System.***out***.println("Obj2 hashCode:"+obj2.hashCode());

System.***out***.println("Obj2 hashCode:"+obj3.hashCode()+"\n");

Student1 s1=**new** Student1(5,"Kedar");

Student1 s2=s1;

System.***out***.println("Hash Code of S1:"+s1.hashCode());

System.***out***.println("Hash Code of S1:"+s2.hashCode());

}

}

**class** Student1 {

**int** id;

String name;

**public** Student1(**int** id, String name) {

**this**.name = name;

**this**.id = id;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **boolean** equals(Object s1) {

System.***out***.println("In equals method of Student class");

**if**(s1==**null**)

**return** **false**;

**if**(!(s1 **instanceof** Student1))

**return** **false**;

Student1 s = (Student1)s1;

**if**(**this**.id==s.getId() && **this**.name==s.getName())

**return** **true**;

**else**

**return** **false**;

}

/\* If we don't implement hash code then even if Objects are equal in nature

\* their hashcode will not be same(not satisfying the contract)

\*/

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

**return** **this**.id \* 25 + **this**.name.length();

}

/\* Remember

1) If two objects are equal, then they must have the same hash code.

2) If two objects have the same hash code, they may or may not be equal.

\*/

}

**Example2:**

**Contract of equals() and hashCode()**

If two objects are equal according to the *equals(Object)* method, then calling the *hashcode()* method on each of the two objects must produce the same integer result.

public class Student {

private int id;

private String name;

public Student(int id, String name) {

this.name = name;

this.id = id;

}

​public int getId() { return id; }

public void setId(int id) { this.id = id; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

}

public class HashcodeEqualsTest {

public static void main(String[] args) {

Student s1 = new Student(1,"Kunal");

Student s2 = new Student(1,"Kunal");

System.out.println("S1 hashcode = "+ s1.hashCode());

System.out.println("S2hashcode = "+ s2.hashCode());

System.out.println("Equality between s1 and s2= "+ s1.equals(s2));

}

}

/\*

Output

S1 hashcode = 1993134103

S2 hashcode = 604107971

Equality between s1 and s2= false

comment : If you want to get the equals and hashCode() contract true then we need to implement equals and hascode

method.

\*/

# Strings In Java

* A string is a sequence of characters. For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'.
* Use **double quotes** to represent a string in Java.

## **String Class**

How to create a string object?

There are two ways to create String object:

* By string literal e.g : String str1 = "Hello Java";
* By new keyword e.g : String str2 = new String("Hello Java");

Note: all string variables are instances of the String class (are not primitive types (like int, char, etc))

**String Class declaration in Java**

public final class String extends Object implements Serializable, Comparable<String>, CharSequence

{

}

**Immu**

**String program 1.**

class StringMain {

public static void main(String[] args) {

// create strings

String s1 = "Java";

String s2 = "CPP";

String s3 = "Angular";

// print strings

System.out.println(s1);

System.out.println(s2);

System.out.println(s3);

}

}

**String Program 2:**

package com.corejava.learning;

public class FirstStringProgram {

public static void main(String args[]) {

System.out.println("First String program");

// 1. way to create a string - literal way

String s1="Hello All";

System.out.println(s1);

//2. way to create a string - new operator

String s2=new String("Hello Java");

System.out.println(s2);

System.out.println("Length of S1 :" + s1.length());

System.out.println("Length of S2 :" + s2.length());

System.out.println("S1 toUpper:" +s1.toUpperCase());

System.out.println("S1 toLower:" +s1.toLowerCase());

// Each string is internally an array.

System.out.println(s1.charAt(0));

System.out.println(s1.charAt(6));

}

}

### **Methods of String.**

|  |  |  |
| --- | --- | --- |
| No. | Method | Description |
| 1 | char charAt(int index) | returns char value for the particular index |
| 2 | int length() | returns string length |
| 3 | static String format(String format, Object... args) | returns a formatted string. |
| 4 | static String format(Locale l, String format, Object... args) | returns formatted string with given locale. |
| 5 | String substring(int beginIndex) | returns substring for given begin index. |
| 6 | String substring(int beginIndex, int endIndex) | returns substring for given begin index and end index. |
| 7 | [boolean contains(CharSequence s)](https://www.javatpoint.com/java-string-contains) | returns true or false after matching the sequence of char value. |
| 8 | [static String join(CharSequence delimiter, CharSequence... elements)](https://www.javatpoint.com/java-string-join) | returns a joined string. |
| 9 | [static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)](https://www.javatpoint.com/java-string-join) | returns a joined string. |
| 10 | boolean equals(Object another) | checks the equality of string with the given object. |
| 11 | boolean isEmpty() | checks if string is empty. |
| 12 | String concat(String str) | concatenates the specified string. |
| 13 | String replace(char old, char new) | replaces all occurrences of the specified char value. |
| 14 | String replace(CharSequence old, CharSequence new) | replaces all occurrences of the specified CharSequence. |
| 15 | static String equalsIgnoreCase(String another) | compares another string. It doesn't check case. |
| 16 | String[] split(String regex) | returns a split string matching regex. |
| 17 | String[] split(String regex, int limit) | returns a split string matching regex and limit. |
| 18 | String intern() | returns an interned string. |
| 19 | int indexOf(int ch) | returns the specified char value index. |
| 20 | int indexOf(int ch, int fromIndex) | returns the specified char value index starting with given index. |
| 21 | int indexOf(String substring) | returns the specified substring index. |

One of the most important characteristics of a string in Java is that they are immutable. In other words, once created, the internal state of a string remains the same throughout the execution of the program. This immutability is achieved through the use of a special string constant pool in the heap.

**SCP: String constant pool** or String literal Pool.

Literal: Things which doesn’t change. E.g 10, 120, 3.145, “Apple”, “Hello” etc.

String literal in Java is created by using double quotes.

For example:

String s="Hello"

The string literal is always created in the string constant pool. In Java, String constant pool is a special area that is used for storing string objects.

Internally, String class uses a string constant pool (SCP). This SCP area is part of the method area (**Permgen**) until Java 1.6 version.

From Java 1.7 onwards, **SCP area is moved in the heap memory** because SCP is a fixed size in the method area but in the heap memory, SCP can be expandable.

Therefore, the string constant pool has been moved to heap area for memory utilization only.

Whenever we create a string literal in Java, JVM checks string constant pool first. If the string already exists in string constant pool, no new string object will be created in the string pool by JVM.

Example

//creates two objects. One is in heap because of new keyword and another one is in SCP for future use.

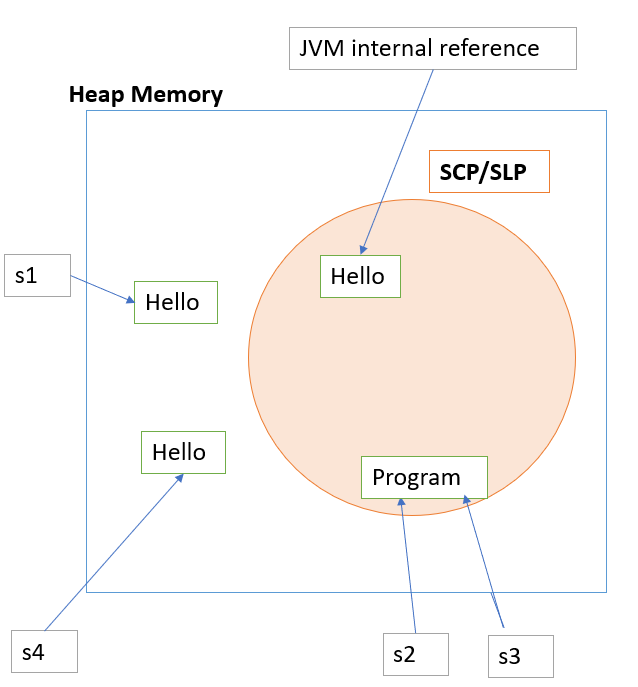
String s1=new String("Hello")

//creates only one object, in SCP

String s2="Program"; // Literal (constant/doesn't change)

String s3="Program"; // Literal – Does not create any Object as literal is already present in SCP.

String s4=new String("Hello"); // will create string Object in heap.



Thus, its preferable to create a string without new as you would have less string objects in memory.

String s4="Java"

String objects those are available in SCP are not applicable for Garbage collection as reference is maintain by JVM.

Interview Question.

String s = new string ("ABC"). How many objects are created after the above statement?

* So, when the class containing this code is loaded, JVM finds the string literal "ABC", creates a string object and puts its reference into string constant pool.

Now when the statement is executed, JVM is forced to create a new string object with contents "ABC" because of the new operator. This time no reference will be put into string constant pool because already there is reference for the same literal.

* So, two objects are created by the statement.

**Key points:**

1. Strings are objects in Java.

2. When we create a String object, it cannot be changed further. In other words, once a String object has been created, we cannot change any characters in string. Therefore, string object is immutable in Java.

4. String class in Java has numerous methods for string manipulation like length() will return the total number of characters in the string.

5. String is not a primitive data type. It is a reference data type.

### **Immutable String**

**Java Strings are Immutable**

In Java, strings are **immutable**. This means, once we create a string, we cannot change that string.

// create a string

String str1 = "Hello! ";

str1 = str1 + "Hi ";

Let's see what has happened here,

* JVM takes the first string "Hello! "
* creates a new string by adding "Hi" to the first string
* assign the new string "Hello! Hi" to the str1 variable
* the first string "Hello! " remains unchanged in SCP

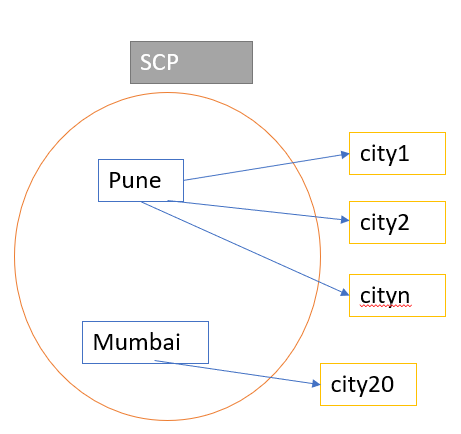
**Why Strings are immutable**

Let’s suppose we have huge number of person objects, and each person has city associated.

String city1="Pune"

String city2="Pune"

String city3="Pune"

. 

String cityn="Pune"

String city20="Mumbai"

Thus, why to create multiple string with value "Pune". Create single string and make it immutable.

**Why Strings are immutable or final in Java.**

There are several benefits of String because it’s immutable and final.

* String Pool is possible because String is immutable in java.
* It increases security because any hacker can’t change its value and it’s used for storing sensitive information such as database username, password etc.
* Since String is immutable, it’s safe to use in multi-threading and we don’t need any synchronization.
* Strings are used in java classloader and immutability provides security that correct class is getting loaded by Classloader.

### **String comparison**

There are three ways to compare String in Java:

* By Using equals() Method :- It compares values of string for equality.(string class has overridden equals() method.
* By Using == Operator:- The == operator compares references not values.
* By compareTo() Method :-

The String class compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

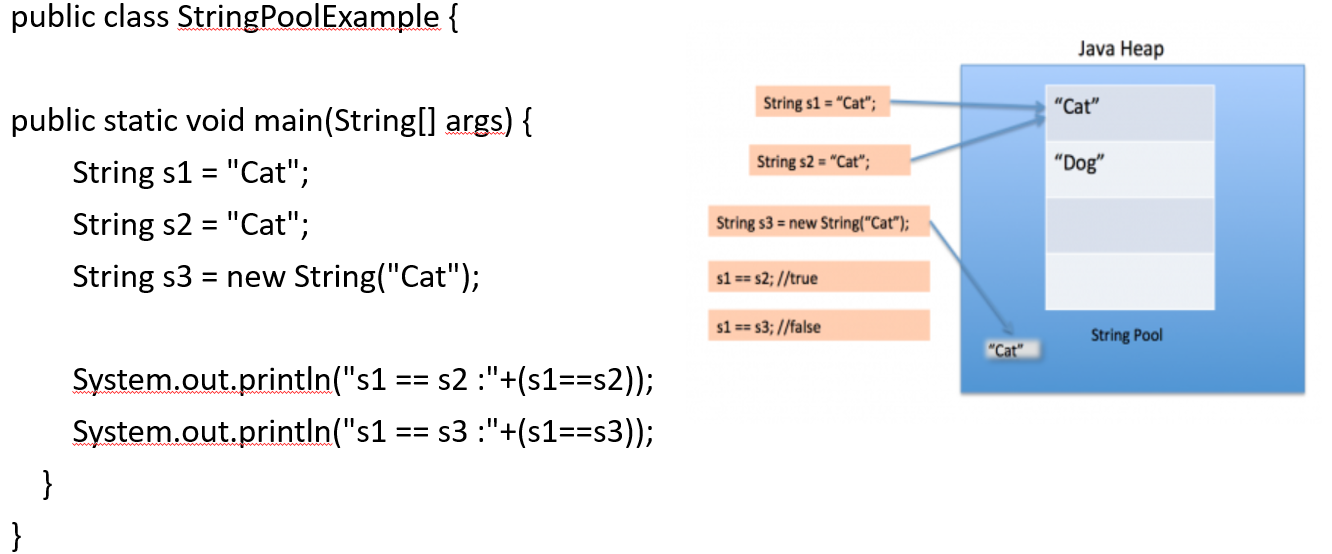
Suppose s1 and s2 are two String objects. If:

**s1 == s2** : The method returns 0.

**s1 > s2** : The method returns a positive value. "Hello" > "Aello"

**s1 < s2** : The method returns a negative value.

**Another example of String pool**



## **String buffer**

StringBuffer is a peer class of String that provides much of the functionality of strings. String represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences.

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| StringBuffer() | It creates an empty String buffer with the initial capacity of 16. |
| StringBuffer(String str) | It creates a String buffer with the specified string.. |
| StringBuffer(int capacity) | It creates an empty String buffer with the specified capacity as length. |

StringBuffer may have characters and substrings inserted in the middle or appended to the end. It will automatically grow to make room for such additions and often has more characters pre-allocated than are actually needed, to allow room for growth.

**package** com.learning.corejava.basics;

**public** **class** StringBufferExample {

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

// Its a thread-safe

StringBuffer sb=**new** StringBuffer("Hello SB"); // capacity 16 + length of parameter

System.***out***.println("Capacity :"+sb.capacity());

sb.append("I am learning Java"); // increase the capacity by (current\*2)+2.

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

sb.insert(8, "!!"); // You can insert substring into the string usig StringBuffer

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

System.***out***.println("Capacity :"+sb.capacity());

}

}

**What is the difference between the String and StringBuffer?**

The String is a final class in Java. The String is immutable. That means we can not change the value of the String object afterword.

Since the string is widely used in applications, we have to perform several operations on the String object, which generates a new String object each time, and all previous objects will be garbage object putting the pressure on the Garbage collector.

Hence, the Java team introduced the StringBuffer class. It is a mutable String object, which means you can change its value.

The string is immutable, but the StringBuffer is mutable.

**Difference between the StringBuffer and StringBuilder?**

|  |  |
| --- | --- |
| **StringBuffer** | **StringBuilder** |
| StringBuffer operations are thread-safe and synchronized | StringBuilder operations are not thread-safe are not-synchronized. |
| StringBuffer is to used when [multiple threads](https://www.edureka.co/blog/java-thread/) are working on the same String | StringBuilder is used in a single-threaded environment. |
| StringBuffer performance is slower when compared to StringBuilder | StringBuilder performance is faster when compared to StringBuffer |
| Syntax: StringBuffer var = new StringBuffer(str); | Syntax: StringBuilder var = new StringBuilder(str); |

**String, StringBuffer and StringBuilder.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Parameter** | **String** | **StringBuffer** | **StringBuilder** |
| **Storage** | String Pool | Heap | Heap |
| **Mutability** | Immutable | Mutable | Mutable |
| **Thread Safe** | Not used in a threaded environment | Used in a multi-threaded environment | Used in a single-threaded environment |
| **Performance** | Slow | Slower than StringBuilder but faster than String | Faster than StringBuffer |
| **Syntax** | String var =“Java”;  String var=new String(“Java”); | StringBuffer var = new StringBuffer("Java"); | StringBuilder var = new StringBuilder("Java"); |

# Interface

An interface in Java is a mechanism that is used to achieve complete abstraction. It is basically a kind of class that contains only constants and abstract methods.

* Data members declared in an interface are by default public, static, and final.
* Only abstract and public modifiers are allowed for methods in interfaces
* We cannot create an object of interface using new operator. But we can create a reference of interface type and interface reference refers to objects of its implementation classes
* From Java 8 onwards, we can define static and default methods in an interface. Prior to Java 8, it was not allowed.
* A variable in an interface must be initialized at the time of declaration.

# Comparator interface

The Java Comparator interface, java.util.Comparator, represents a component that can compare two objects so they can be sorted using sorting functionality.

The Java Comparator interface definition looks like this:

public interface Comparator<T> {

public int compare(T o1, T o2);

}

Notice that the Java Comparator interface only has a single method. This method, the compare() method, takes two objects which the Comparator implementation is intended to compare. The compare() method returns an int which signals which of the two objects was larger. The semantics of the return values are:

A negative value means that the first object was smaller than second object.

The value 0 means the two objects are equal.

A positive value means that the first object was larger than the second object.

You will find the Example of comparator below in Arraylist

# Collection

Collection (I) : Group of element will be represented by single entity.

It is root interface for all the Collection.

All classes and interfaces are part of java.util package.

Common methods applicable for any Collection object:

boolean add(Object o)

boolean isEmpty()

boolean addAll(Collection c)

boolean contains(Object o)

boolean remove()

boolean containsAll(Collection c)

void clear()

Object[] toArray()

boolean retainAll(Collection c)

Iterator iterator()

boolean removeAll(Collection c)

int size()

# **List interface**

In Java, the **List** interface is an ordered collection that allows us to store and access elements sequentially. It extends the Collection interface.

Lists typically allow duplicate elements

List is an interface; we cannot create objects from it.

Features of the List

1. The list allows storing duplicate elements in Java. JVM differentiates duplicate elements by using ‘index’ (Position). It always starts at zero.

2. In the list, we can add an element at any position.

3. It maintains insertion order. i.e., List can preserve the insertion order by using the index.

4. It allows for storing many null elements.

5. Java list uses a resizable array for its implementation. Resizable means size can grow, or we can increase or decrease the size of the array.

6. Except for LinkedList, ArrayList, and Vector is an indexed-based structure.

7. It provides a special Iterator called a ListIterator that allows accessing the elements in the forward direction using hasNext() and next() methods.

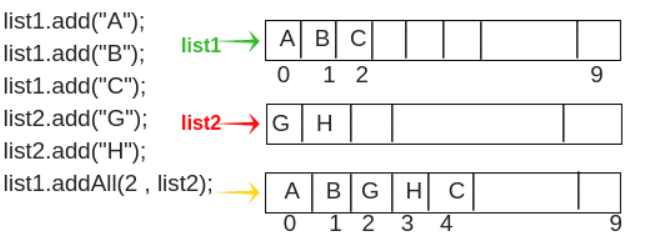
In order to use functionalities of the List interface, we can use these classes:

* [ArrayList](https://www.programiz.com/java-programming/arraylist)
* [LinkedList](https://www.programiz.com/java-programming/linkedlist)
* [Vector](https://www.programiz.com/java-programming/vector)
* [Stack](https://www.programiz.com/java-programming/stack)

## **ArrayList**

The ArrayList class is a resizable [array](https://www.w3schools.com/java/java_arrays.asp), which can be found in the java.util package.

The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an ArrayList whenever you want.



**Where does ArrayList store its element**

Internally it uses array to store its element. It’s an Object array which is defined as follows.

transient Object[] elementData;

**Constructor**

Initial capacity of the created ArrayList depends on the constructor used.

* ArrayList(int initialCapacity)– If initial capacity is explicitly specified while constructing an ArrayList then it is ensured that the elementData array is created with that length.

Internal java Code:

*this.elementData = new Object[initialCapacity];*

* ArrayList()– If no initial capacity is specified then the ArrayList is created with the default capacity 10

Internal java Code:

*private static final int DEFAULT\_CAPACITY = 10;*

*public ArrayList() {*

*this.elementData = DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA;*

*}*

**Basic ArrayList Example**

package com.corejava.itp.collection;

import java.util.ArrayList;

import java.util.List;

import java.util.Vector;

/\*

\* Resizable-array implementation of the List interface.

\*/

public class ArrayListBasics {

public static void main(String[] args) {

// Integer a=10; // primitive to wrapper.

ArrayList myList = new ArrayList(); // default capacity(length or size) is 10

myList.add(10); // int 10 will be converted to Integer wrapper class.

myList.add(20);

myList.add(30);

myList.add(40);

myList.add(50);

myList.add(60);

myList.add(70);

myList.add(80);

myList.add(90);

myList.add(100); // size/capacity will be increase

myList.add(110);

myList.add("ONE");

myList.add("Two");

myList.add("Three");

System.out.println(myList);

System.out.println("ArrayList size :"+myList.size());

System.out.println("Element at index 0:"+myList.get(0)); // array[0]

System.out.println("Element at index 2:"+myList.get(2)); // array[2]

System.out.println("Element at index 10:"+myList.get(10)); // array[10]

//System.out.println("Element at index 50:"+myList.get(50)); // array[10]

System.out.println("Does ArrayList contains 30??: "+myList.contains(30));

System.out.println("Does ArrayList contains 300??: "+myList.contains(300));

System.out.println("Remove element at index 5:"+myList.remove(5));

System.out.println("ArrayList after remove :"+myList);

System.out.println("ArrayList travel : ");

for(int i=0;i<myList.size();i++) {

System.out.println(myList.get(i));

}

// Define the ArrayList with specific type of element. E.g. If ArrayList will have only int.

// Below ArrayList will hold only integer.

ArrayList<Integer> intList = new ArrayList<Integer>();

intList.add(101);

intList.add(102);

intList.add(103);

intList.add(104);

//print ArrayList using for each

System.out.println("\nInteger arraylist printed using for-each loop");

for(Integer i:intList) {

System.out.println(i);

}

ArrayList<Integer> studRoll=new ArrayList<Integer>();

studRoll.addAll(intList);

studRoll.add(105);

studRoll.add(106);

System.out.println("Stud Roll Numbers :"+studRoll);

// ArrayList Constructor

ArrayList<String> strList = new ArrayList<String>(); // Initial size will be 10

ArrayList<String> strList1 = new ArrayList<String>(100); // Initial size/capacity will be 100

//Another way to define the ArrayList using interface reference.

// Imp: We can't create object of an interface

//List aList=new List(); //not valid- should not create interface object.

List aList= new ArrayList<Integer>();

/\*

\* Difference between below statements

ArrayList<Integer> intList = new ArrayList<Integer>();

List aList= new ArrayList<Integer>();

intList can't be converted to other list implementation type.

aList can be converted to other List implementation type like ArrayList, Vector and LinkedList

\*/

aList = new Vector<Integer>();

// intList = new Vector<Integer>(); //error in this list as intList cant convert to Vector.

}

}

Advance Example of ArrayList

package com.corejava.collections.learning;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.Iterator;

import java.util.List;

public class ArrayListDemo {

public static void main(String[] args) {

ArrayList aList = new ArrayList();

//aList = new Vector(); you can't covert it to Vector as aList is declared as ArrayList.

// Below declaration is more suitable, Program to interface and not to implementation

// i.e create a reference of supertype and assign memory of subtype.

List aList1 = new ArrayList();

// ArrayList with only integer object. Initial capacity is 10

ArrayList<Integer> intList = new ArrayList<Integer>();

// ArrayList with String object with initial capacity 20

ArrayList<String> strList = new ArrayList<String>(20);

aList.add("Zero");

aList.add("One");

aList.add("two");

aList.add(3);

aList.add("Four");

System.out.println("0th Element of aList using get(index):"+aList.get(0));

System.out.println("Complete array List :"+aList);

regularforWay(aList);

forEachWay(aList);

iteratorWay(aList);

System.out.println("\nArrayList contains method, check for Four: "+aList.contains("Four"));

System.out.println("\nArrayList remove with index method, remove 3: "+aList.remove(3));

System.out.println("\nArrayList set (replace) with index method: "+aList.set(2,"twoo"));

strList.add("Five");

strList.add("Seven");

strList.add("One");

strList.add("Four");

strList.add("Nine");

sortList(strList);

sortListInReverse(strList);

customSort(strList);

ArrayList<String> revList = reverseArrayList(strList);

}

static void regularforWay(List a) {

int i=0;

System.out.println("\nArraylist using regular for loop");

for(i=0;i<a.size();i++) {

System.out.println(a.get(i));

}

}

static void forEachWay(List a) {

System.out.println("\nArraylist using foreach loop\n");

for(Object s:a) {

System.out.println(s);

}

}

static void iteratorWay(List a) {

System.out.println("\nArraylist using iterator\n");

Iterator it=a.iterator();

while(it.hasNext()) {

System.out.println(it.next());

}

}

static void sortList(List<String> aList) {

System.out.println("\nSort using Collections");

System.out.println("Before :"+aList);

Collections.sort(aList);

System.out.println("After :"+aList);

}

static void sortListInReverse(List<String> aList) {

System.out.println("\nReverse Sort using Collections");

System.out.println("Before :"+aList);

Collections.sort(aList,Collections.reverseOrder());

System.out.println("After :"+aList);

}

static void customSort(List<String> aList) {

// Implement comparator interface

System.out.println("\nSort using Comparator");

System.out.println("Before:"+aList);

Collections.sort(aList,new MySortComparator());

System.out.println("After:"+aList);

}

static ArrayList<String> reverseArrayList(ArrayList<String> alist) {

// ArrayList for storing reversed elements

ArrayList<String> revArrayList = new ArrayList<String>();

for (int i = alist.size() - 1; i >= 0; i--) {

// Append the elements in reverse order

revArrayList.add(alist.get(i));

}

// Return the reversed arraylist

return revArrayList;

}

}

class MySortComparator implements Comparator<String>{

@Override

public int compare(String o1, String o2) {

// o1>o1 it will return positive

// o1==o2 it will return zero

// o1<o2 return negative

return o1.compareTo(o2);

}

}

**Another Example of ArrayList with Comparator implementation.**

package com.corejava.collections.learning;

import java.lang.\*;

import java.util.\*;

class Student {

int rollno;

String name, address;

// Constructor

public Student(int rollno, String name, String address) {

this.rollno = rollno;

this.name = name;

this.address = address;

}

// Used to print student details in main()

public String toString()

{

return this.rollno + " " + this.name + " " + this.address;

}

}

class Sortbyroll implements Comparator<Student> {

// Used for sorting in ascending order of roll number

public int compare(Student a, Student b) {

return a.rollno - b.rollno;

}

}

class Sortbyname implements Comparator<Student> {

// Used for sorting in ascending order of name

public int compare(Student a, Student b) {

return a.name.compareTo(b.name);

}

}

// Main class

public class ArrayListCustomSort {

public static void main(String[] args) {

ArrayList<Student> ar = new ArrayList<Student>();

ar.add(new Student(601, "Ram", "pune"));

ar.add(new Student(571, "Kunal", "delhi"));

ar.add(new Student(491, "Samar", "jaipur"));

ar.add(new Student(491, "Samir", "jaipur"));

System.out.println("Unsorted");

for (int i = 0; i < ar.size(); i++)

System.out.println(ar.get(i));

Collections.sort(ar, new Sortbyroll());

System.out.println("\nSorted by rollno");

for (int i = 0; i < ar.size(); i++)

System.out.println(ar.get(i));

Collections.sort(ar, new Sortbyname());

System.out.println("\nSorted by name");

for (int i = 0; i < ar.size(); i++)

System.out.println(ar.get(i));

}

}

/\* Output

Unsorted

601 Ram pune

571 Kunal delhi

491 Samar jaipur

491 Samir jaipur

Sorted by rollno

491 Samar jaipur

491 Samir jaipur

571 Kunal delhi

601 Ram pune

Sorted by name

571 Kunal delhi

601 Ram pune

491 Samar jaipur

491 Samir Jaipur \*/

**Interview question –**

1. reverse the list without using temp list

**package** com.learning.corejava.collection;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**public** **class** ReverseArrayList {

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

ArrayList<Integer> a=**new** ArrayList<Integer>(Arrays.*asList*(1,3,4,6,7,8,10));

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

**for**(**int** i=0;i<a.size();i++) {

a.add(i, a.remove(a.size()-1));

}

}

}

1. How does add/remove method work in ArrayList

**Answer:**

ArrayList is internally implemented as growable or resizable Array.

If you see the ArrayList internal implementation in Java, everytime add() method is called it is ensured that ArrayList has required capacity.

If the capacity is exhausted a new array is created with 50% more capacity than the previous one. All the elements are also copied from the previous array to the new array.

How does remove method work in ArrayList.

If you remove any element from an array then all the subsequent elements are to be shifted to fill the gap created by the removed element.

## **LinkedList**

Java LinkedList class uses a doubly linked list to store the elements.

Features

* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* Java LinkedList class is non synchronized.
* In Java LinkedList class, manipulation is fast because no shifting needs to occur.
* Java LinkedList class can be used as a list, stack or queue.

LinkedList constructor and implementation

|  |  |
| --- | --- |
| LinkedList() | It is used to construct an empty list. |
| LinkedList(Collection<? extends E> c) | It is used to construct a list containing the elements of the specified collection, in the order, they are returned by the collection's iterator. |

//Definition of LinkedList class

public class LinkedList<E> implements List<E>, Deque<E>, Cloneable, Serializable

{

}

**package** com.corejava.collections.learning;

**import** java.util.\*;

**public** **class** LinkedListDemo{

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

LinkedList<String> langList=**new** LinkedList<String>();

langList.add("Java");

langList.add("CPP");

langList.add("Python");

langList.add("R-lang");

System.***out***.println("Print the newly created LinkedList:");

Iterator<String> itr=langList.iterator();

**while**(itr.hasNext()){

System.***out***.println(itr.next());

}

// Adding an element at the specific position

langList.add(1, "React");

System.***out***.println("After add(int index, E element) method: "+langList);

// Adding an element at the first position

langList.addFirst("Angular");

System.***out***.println("After invoking addFirst(E e(i.e 'Angular')) method: "+langList);

//Adding an element at the last position

langList.addLast("AWS-Cloud");

System.***out***.println("After invoking addLast(E e(i.e 'AWS-Cloud')) method: "+langList);

//Removing specific element from arraylist

langList.remove("CPP");

System.***out***.println("After invoking remove(object) method-removed 'CPP': "+langList);

//Removing element on the basis of specific position

langList.remove(0);

System.***out***.println("After invoking remove(index) method- removed at index 0: "+langList);

langList.removeFirst();

System.***out***.println("After invoking removeFirst() method"+langList);

langList.removeLast();

System.***out***.println("After invoking removeLast() method"+langList);

langList.add("Java-Spring");

langList.add("Java-Spring");

langList.add("Java-Spring");

langList.add("Java-Spring");

System.***out***.println("After invoking add('Java\_Spring') method 4 times"+langList);

langList.remove("Java-Spring");

System.***out***.println("After invoking remove('Java\_Spring') method:"+langList);

langList.removeFirstOccurrence("Java-Spring");

System.***out***.println("After invoking removeFirstOcurrence() method"+langList);

langList.removeLastOccurrence("Java-Spring");

System.***out***.println("After invoking removeLastOcurrence() method"+langList);

System.***out***.println("Print the LinkedList in reverse Order");

*reverseLinkedList*(langList);

}

**static** **void** reverseLinkedList(LinkedList langList) {

Iterator itr =langList.descendingIterator();

**while**(itr.hasNext()){

System.***out***.println(itr.next());

}

}

}

Apart from regular List methods LinkedList provides few extra methods as below.

|  |  |
| --- | --- |
| Methods | Descriptions |
| addFirst() | adds the specified element at the beginning of the linked list |
| addLast() | adds the specified element at the end of the linked list |
| getFirst() | returns the first element |
| getLast() | returns the last element |
| removeFirst() | removes the first element |
| removeLast() | removes the last element |
| peek() | returns the first element (head) of the linked list |
| poll() | returns and removes the first element from the linked list |
| offer() | adds the specified element at the end of the linked list |

**How does LinkedList class store its element?**

Internally LinkedList class in Java uses objects of type **Node** to store the added elements. Node is implemented as a static class with in the LinkedList class.

private static class Node<E> {

E data;

Node<E> next;

Node<E> prev;

Node(Node<E> prev, E element, Node<E> next) {

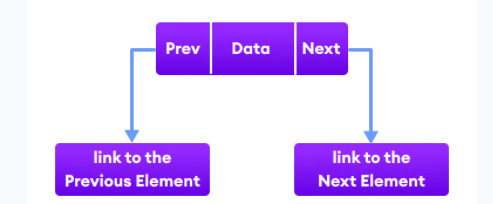
this.item = element;

this.next = next;

this.prev = prev;

}

}



**How add() method works in a LinkedList?**

Since it is a linked list so apart from regular **add()** method to add sequentially there are **addFirst()** and **addLast()** methods also in Java LinkedList class.

There are also separate variables for holding the reference of the first and last nodes of the linked list.

/\*\*

\* Pointer to first node.

\*/

transient Node<E> first;

/\*\*

\* Pointer to last node.

\*/

transient Node<E> last;

**Add() in LinkedList**

private void linkFirst(E paramE) {

Node<E> node1 = this.first;

Node<E> node2 = new Node<>(null, paramE, node1);

this.first = node2;

if (node1 == null) {

this.last = node2;

} else {

node1.prev = node2;

}

this.size++;

this.modCount++;

}

Add() element at last position

/\*\*

\* Links e as last element. Called when add() or addLast() is invoked in a program.

\*/

void linkLast(E e) {

final Node<E> l = last;

final Node<E> newNode = new Node<>(l, e, null);

last = newNode;

if (l == null)

first = newNode;

else

l.next = newNode;

size++;

modCount++;

}

### **ArrayList and LinkedList Difference**

**1) Search/Retrieve**

ArrayList search is faster than LinkedList search as ArrayList implement RandomAccess Interface.

get(int index) in ArrayList gives the performance of O(1)

get(int index) in LinkedList performance is O(n).

**2) Insert**

LinkedList add method gives O(1) performance while ArrayList gives O(n) in worst case.

**3) Deletion**

LinkedList remove operation gives O(1) performance while ArrayList gives variable performance: O(n) in worst case (while removing first element) and O(1) in best case (While removing last element)

**4)** **Memory Overhead**

ArrayList maintains indexes & element data while LinkedList maintains element data and two pointers for neighbor nodes hence the memory consumption is high in LinkedList comparatively.

**5) Underline Implementation**

ArrayList : Underline data structure is Resizable and growable Array

LinkedList: Underline data structure is doubly linked list.

**6) Parent interface.**

ArrayList implements RandomAccess interface.

LinkedList does not implement RandomAcess interface.

When to use ArrayList or LinkedList

If there is a requirement of frequent addition and deletion in an application, then LinkedList is a best choice.

Search (get method) operations are fast in Arraylist (O(1)) but not in LinkedList (O(n)) so If there are less add and remove operations and more search operations requirement, ArrayList would be your best

choice.

## **Vector**

The Vector class is an implementation of the List interface that allows us to create resizable-arrays similar to the [ArrayList](https://www.programiz.com/java-programming/arraylist" \o "Java ArrayList) class.

The Vector class synchronizes each individual operation. This means whenever we want to perform some operation on vectors, the Vector class automatically applies a lock to that operation.

Simple example

import java.util.Vector;

class Main {

public static void main(String[] args) {

Vector<String> mammals= new Vector<>();

// Using the add() method

mammals.add("Dog");

mammals.add("Horse");

// Using index number

mammals.add(2, "Cat");

System.out.println("Vector: " + mammals);

// Using addAll()

Vector<String> animals = new Vector<>();

animals.add("Crocodile");

animals.addAll(mammals);

System.out.println("New Vector: " + animals);

}

}

Another example

package com.corejava.collections.learning;

import java.util.Collections;

import java.util.Enumeration;

import java.util.Iterator;

import java.util.Vector;

public class VectorDemo {

public static void main(String[] args) {

Vector<String> langVector=new Vector<String>();

langVector.add("Java");

langVector.add("CPP");

langVector.add("Python");

langVector.add("R-lang");

System.out.println("Print the complete vector object: "+langVector);

System.out.println("Capacity of the vector:"+langVector.capacity());

System.out.println("Does vector has CPP :"+langVector.contains("CPP"));

System.out.println("get the element at index 3:"+langVector.get(3));

System.out.println("get the index of 'Python'"+langVector.indexOf("Python"));

System.out.println("set method to replace java->JAVA"+langVector.set(0, "JAVA"));

// In iterator you can travel the collection as well as remove the elements from it.

// Using Enumeration You can't do the modification(add/remove) inside collection.

// getting the Enumeration object over Vector

Enumeration enumeration = Collections.enumeration(langVector);

System.out.println("printing each enumeration constant by enumerating through the Vector:");

while (enumeration.hasMoreElements()) {

System.out.println(enumeration.nextElement());

}

}

}

## **Stack**

The Java collections framework has a class named Stack that provides the functionality of the stack data structure.

The Stack class extends the Vector class.

In order to create a stack, we must import the java.util.Stack package first. Once we import the package, here is how we can create a stack in Java.

Stack<Type> stacks = new Stack<>();

Here, Type indicates the stack's type. For example,

// Create Integer type stack

Stack<Integer> stacks = new Stack<>();

// Create String type stack

Stack<String> stacks = new Stack<>();

Sample Code

import java.util.Stack;

class Main {

public static void main(String[] args) {

Stack<String> animals= new Stack<>();

// Add elements to Stack

animals.push("Dog");

animals.push("Horse");

animals.push("Cat");

System.out.println("Initial Stack: " + animals);

// Remove element stacks

String element = animals.pop();

System.out.println("Removed Element: " + element);

}

// Access element from the top using peek() method.

String element = animals.peek();

System.out.println("Element at top: " + element);

// Check if stack is empty

boolean result = animals.empty();

System.out.println("Is the stack empty? " + result);

}

Short summary of List

/\*

ArrayList vector and LinkedList difference

Internal logic

ArrayList : Array

LinkedList : doubly Linked list

Vector : Array

Declaration

ArrayList -> ArrayList a=new ArrayList() or List a=new ArrayList();

LinkedList -> LinkedList list=new LinkedList() or List list=new LinkedList();

Vector -> Vector v=new Vector() or List v=new Vector();

Synchronized - main difference

ArrayList : Methods are non Synchronized

LinkedList : Methods are non Synchronized

Vector : Methods are Synchronized means Vectors are thread safe.

When to use what

ArrayList : When retrieve operation is more. Because it has Random access a[5] or a[1000]

LinkedList : Retrieve operations are slow. Use when Insert or remove is frequent operation.

Vector : Use Vector then program is multi-threaded program. When retrieve operation is more. Because it has Random access a[5] or a[1000]

\*/

## **Iterators in Java**

**Iterators in Java** are used to retrieve the elements one by one from a collection object. They are also called cursors in java

There are four types of iterators or cursors available in Java. They are as follows:

* Enumeration
* Iterator
* ListIterator
* Spilterator (Java 1.8)

### **Enumeration in Java**

Enumeration is the first iterator that was introduced in Java 1.0 version. It is located in java.util package. It is a legacy interface that is implemented to get elements one by one from the legacy collection classes such as Vector and Properties.

Legacy classes are those classes that are coming from the first version of Java. Early versions of Java do not include “[collections framework](https://www.scientecheasy.com/2020/09/java-collections-framework.html/)”. Instead, it defined several classes and one interface for storing objects.

When collections came in the Java 1.2 version, several of the original classes were re-engineered to support the collection interfaces.

Thus, they are fully compatible with the framework. These old classes are known as legacy classes. The legacy classes defined by java.util are Vector, Hashtable, Properties, Stack, and Dictionary. There is one legacy interface called Enumeration.

Enumeration is read-only. You can just read data from the vector. You cannot remove it from the vector using Enumeration.

Since enumeration is an interface so we cannot create an object of enumeration directly. We can create an object of enumeration by calling elements() method of the Vector class.

Syntax

public Enumeration elements() // Return type is Enumeration.

For example:

Enumeration e = v.elements(); // Here, v is a vector class object.

Methods of Enumeration

The Enumeration interface defines the following two methods. They are as follows:

**1. public boolean hasMoreElements():** When this method is implemented, hasMoreElements() will return true If there are still more elements to extract and false if all the elements have been enumerated.

**2. public Object nextElement():** The nextElement() method returns next element in the enumeration. It will throw NoSuchElementException when the enumeration is complete.

import java.util.Enumeration;

import java.util.Vector;

public class EnumerationTest

{

public static void main(String[] args)

{

// Create object of vector class without using generic.

Vector v = new Vector();

// Add ten elements of integer type using addElement() method.

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

v.addElement(i);

}

System.out.println(v);//print all elements at a time[0, 1, 2, 3, 4, 5]

// Get elements one by one. So, will require Enumeration concept.

// Create object of Enumeration by calling elements() method of vector class using object reference variable v.

// At the beginning, e (cursor) will point to index just before the first element in v.

Enumeration e = v.elements();

// Checking the next element availability using reference variable e and while loop.

while(e.hasMoreElements())

{

// Moving cursor to next element.

Object o = e.nextElement();

Integer i = (Integer)o; // Here, Type casting is required because the return type of nextElement() method is an object. Therefore, it's compulsory to require type casting.

System.out.println(i);

}

}

}

**Limitation of Enumeration**

There are many limitations of using enumeration interface in java. They are as follows:

1. Enumeration concept is applicable for only legacy class. Hence, it is not a universal cursor.  
2. We can get only read operation by using the enumeration. We cannot perform the remove operation.  
3. We can iterate using enumeration only in the forward direction.  
4. Java is not recommended to use enumeration in new projects.

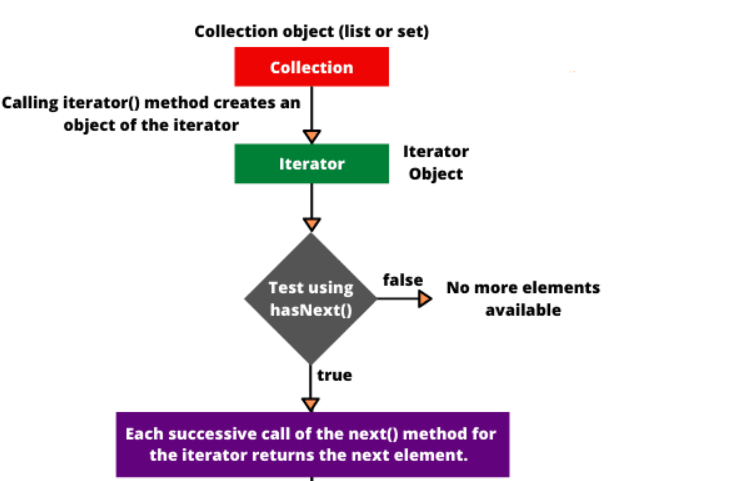
To overcome these limitations, We should go for the next level Iterator concept in Java.

### **Iterator**

Iterator in Java is used in the [Collections Framework](https://www.scientecheasy.com/2020/09/java-collections-framework.html/) to retrieve elements sequentially (one by one). It is called **universal Iterator** or cursors.

It can be applied to any collection object. By using Iterator, we can perform both read and remove operations.

How Java Iterator works internally?



**Difference between Enumeration and Iterator**

Both are useful to retrieve elements from a collection. But the main difference between Enumeration and iterator is with respect to functionality.

By using an enumeration, we can perform only read access but using an iterator, we can perform both read and remove operation.

**Advantage of Iterator in Java**

* An iterator can be used with any collection classes.
* We can perform both read and remove operations.
* It acts as a universal cursor for collection API.

**Limitation of Iterator in Java**

* By using Enumeration and Iterator, we can move only towards forwarding direction. We cannot move in the backward direction. Hence, these are called single-direction cursors.
* We can perform either read operation or remove operation.
* We cannot perform the replacement of new objects.
* For example, suppose there are five Apple in a box. Out of five, two Apple are not good but we cannot replace those damaged one with new Apple.

**Solution is to use ListIterator**

**ListIterator in Java** is the most powerful iterator or cursor that was introduced in Java 1.2 version. It is a bi-directional cursor.

Java ListIterator is an interface (an extension of Iterator interface) that is used to retrieve the elements from a collection object in both forward and reverse directions.

Java ListIterator can be used for all List implemented classes such as ArrayList, CopyOnWriteArrayList, LinkedList, Stack, Vector, etc.

**Methods of ListIterator in Java**

**Forward direction:**

**1. public boolean hasNext():** This method returns true if the ListIterator has more elements when iterating the list in the forward direction.

**2. public Object next():** This method returns the next element in the list. The return type of next() method is Object.

**3. public int nextIndex():** This method returns the index of the next element in the list. The return type of this method is an integer.

**Backward direction:**

**4. public boolean hasPrevious():** It checks that list has more elements in the backward direction. If the list has more elements, it will return true. The return type is boolean.

**5. public Object previous():** It returns the previous element in the list and moves the cursor position backward direction. The return type is Object.

**6. public int previousIndex():** It returns the index of the previous element in the list. The return type is an Integer.

**Other capability methods:**

**7. public void remove():** This method removes the last element returned by next() or previous() from the list. The return type is ‘nothing’.

**8. public void set(Object o):** This method replaces the last element returned by next() or previous() with the new element.

**9. public void add(Object o):** This method is used to insert a new element in the list.

**Example:**

**import java.util.LinkedList;**

**import java.util.List;**

**import java.util.ListIterator;**

**public class ListIteratorTest {**

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

**{**

**List<String> list = new LinkedList<>();**

**list.add("A");**

**list.add("B");**

**list.add("C");**

**// Creating ListIterator object.**

**ListIterator<String> listIterator = list.listIterator();**

**// Traversing elements in forwarding direction.**

**System.out.println("Forward Direction Iteration:");**

**while(listIterator.hasNext()) {**

**System.out.println(listIterator.next());**

**}**

**// Traversing elements in the backward direction. The ListIterator cursor is at just after the last element.**

**System.out.println("Backward Direction Iteration:");**

**while(listIterator.hasPrevious()) {**

**System.out.println(listIterator.previous());**

**}**

**}**

**}**

**Output:**

**Forward Direction Iteration:**

**A**

**B**

**C**

**Backward Direction Iteration:**

**C**

**B**

**A**

ListIterator is the most powerful cursor but it still has some limitations. They are as follows:

1. Java List Iterator is applicable only for list implemented class objects. Therefore, it is not a universal Java cursor.
2. It is not applicable to whole collection API.

|  |  |
| --- | --- |
| **Iterator** | **ListIterator** |
| 1. Java Iterator is applicable to the whole Collection API. | 1. Java ListIterator is only applicable for List implemented classes such as ArrayList, CopyOnWriteArrayList, LinkedList, Stack, Vector, etc. |
| 2. It is a Universal Iterator. | 2. It is not a Universal Iterator in Java. |
| 3. Iterator supports only forward direction Iteration. | 3. ListIterator supports both forward and backward direction iterations. |
| 4. It is known as a uni-directional iterator. | 4. It is also known as bi-directional iterator. |
| 5. Iterator supports only read and delete operations. | 5. ListIterator supports all the operations such as read, remove, replacement, and the addition of the new elements. |
| 6. We can get the Iterator object by calling iterator() method. | 6. We can create ListIterator object by calling listIterator() method. |

**Differences Between Enumeration and Iterator In Java**

**1) Introduction**

***Iterator* interface is introduced from JDK 1.2 where as *Enumeration* interface is there from JDK 1.0.**

**2) remove() method**

***Enumeration* only traverses the *Collection*object. You can’t do any modifications to *Collection* while traversing the *Collection* using *Enumeration*. Where as *Iterator* interface allows us to remove an element while traversing the *Collection*object.**

***Iterator* has *remove()* method which is not there in the *Enumeration* interface.**

**3) Fail-Fast Vs Fail-Safe**

***Iterator* is a fail-fast in nature. i.e it throws *ConcurrentModificationException* if a collection is modified while iterating other than it’s own *remove()* method. Where as *Enumeration* is fail-safe in nature. It doesn’t throw any exceptions if a collection is modified while iterating.**

# **Set**

## **What is Set**

Java Set is a collection of elements (objects) that contains **no duplicate** elements.

Java Set is an interface that extends Collection interface.

Unlike List, Java Set is NOT an ordered collection, it’s elements does NOT have a particular order.

Java Set does **NOT** provide a control over the **position** where you can insert an element. You cannot access elements by their index and also search elements in the list.

Set allows you to add at most one null element only.

Collection(I)

HashSet

Set(I)

TreeSet

LinkedHashSet

SortedSet(I)

Navigable(I)

## **Set Methods**

add() - adds the specified element to the set

addAll() - adds all the elements of the specified collection to the set

iterator() - returns an iterator that can be used to access elements of the set sequentially

remove() - removes the specified element from the set

removeAll() - removes all the elements from the set that is present in another specified set

retainAll() - retains all the elements in the set that are also present in another specified set

clear() - removes all the elements from the set

size() - returns the length (number of elements) of the set

toArray() - returns an array containing all the elements of the set

contains() - returns true if the set contains the specified element

containsAll() - returns true if the set contains all the elements of the specified collection

hashCode() - returns a hash code value (address of the element in the set)

## **HashSet**

* Based on Hash Map
* Duplicate not allowed (try to add duplicate you will get false in return and object wont be added)
* Insertion order not preserved : All objects will be inserted based on Hashcode
* Null allowed
* Search operation will be easy as hashcode will be used.
* It is much faster because of the use of hashing technique and gives constant-time performance for adding (insertion), retrieval, removal, contains, and size operations.
* HashSet class is not synchronized, which means it is not thread-safe. If you want to synchronize HashSet, use Collections.synchronizeSet() method.

Constructor (For all Hash data structure constructors all common)

1. HashSet h= new HashSet() //default capacity 16 and fill ratio/load factor 0.75

After filling 75% of data new hashset object is going to create.

1. HashSet h= new HashSet(int capacity) // load factor 0.75
2. HashSet h= new HashSet(int capacity, float loadFactor)
3. HashSet h= new HashSet(Collection c)

### **HashSet Code Example**

**package** com.learning.corejava.collection;

**import** java.util.\*;

**public** **class** HashSetExample {

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

**return** 10;

}

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

HashSetExample h=**new** HashSetExample();

System.***out***.println("Hashcode of HashSetExample object h:"+h.hashCode());

HashSetExample h1=**new** HashSetExample();

System.***out***.println("Hashcode of HashSetExample object h1:"+h1.hashCode());

// Creating a HashSet

Set<String> courses = **new** HashSet<>();

// Adding new elements to the HashSet

courses.add("Big Data");

courses.add("Node.js");

courses.add("Java");

courses.add("Python");

courses.add("JavaScript");

courses.add("AWS");

courses.add("Machine Learning");

//courses.add(null); // hash code will be zero.null not allowed in TreeSet

//the hash code of a null element is defined to be zero.

System.***out***.println("HashCode of HashSet object :"+courses.hashCode());

// Adding duplicate elements will be ignored and return false.

courses.add("Java");

System.***out***.println("Courses HashSet:"+courses);

// Check if the HashSet contains any specific element

String myCourse = "Node1.js";

**if**(courses.contains(myCourse)) {

System.***out***.println(myCourse + " is in the courses list.");

} **else** {

System.***out***.println(myCourse + " is not in the courses list.");

}

System.***out***.println("Sorting Courses using List.");

List<String> list = **new** ArrayList<String>(courses);

System.***out***.println("List without sort :"+list);

// sort method take list as a parameter. Thus we need to convert HashSet into List(ArrayList)

Collections.*sort*(list);

// Printing the sorted elements of the HashSet

System.***out***.println("Printing the Courses in sorted order using List: " + list);

Collections.*reverse*(list);

System.***out***.println("Printing the Courses in reverse order using List: " + list);

System.***out***.println("Removing items from HashSet using remove()");

courses.remove("Python");

// Iterating over HashSet items

System.***out***.println("\nIterating over course list after removing Python:");

Iterator<String> i = courses.iterator();

**while** (i.hasNext())

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

// Creating another object of HashSet

HashSet<String> eduNewCourses = **new** HashSet<String>();

eduNewCourses.add("Node.js");

eduNewCourses.add("Python");

eduNewCourses.add("Machine Learning");

System.***out***.println("\nCourses Hashset :"+courses);

System.***out***.println("New Courses Hashset :"+eduNewCourses);

//Removing all the new elements from HashSet

System.***out***.println("\nRemove NewCourses hashset from main courses hashset");

courses.removeAll(eduNewCourses);

System.***out***.println("After invoking removeAll() method courses left: "+ courses);

//Removing elements on the basis of specified condition

courses.removeIf(str->str.contains("Java"));

System.***out***.println("After invoking removeIf(Java) method: "+ courses);

System.***out***.println("\nAdd Python and Node.js again in the courses");

courses.add("Node.js");

courses.add("Python");

System.***out***.println("Updated Courses hashset:"+courses);

// Removing elements from eduCourses which are specified in eduNewCourses. This is a kind of intersection of two set.

courses.retainAll(eduNewCourses);

System.***out***.println("\nHashSet after " + "retainAll() operation : " + courses);

//Removing all the elements available in the set

courses.clear();

System.***out***.println("After invoking clear() method: "+ courses);

}

}

**Output**:

Hashcode of HashSetExample object h:10

Hashcode of HashSetExample object h1:10

HashCode of HashSet object :2025757310

Courses HashSet:[Java, JavaScript, Big Data, Node.js, AWS, Machine Learning, Python]

Node1.js is not in the courses list.

Sorting Courses using List.

List without sort :[Java, JavaScript, Big Data, Node.js, AWS, Machine Learning, Python]

Printing the Courses in sorted order using List: [AWS, Big Data, Java, JavaScript, Machine Learning, Node.js, Python]

Printing the Courses in reverse order using List: [Python, Node.js, Machine Learning, JavaScript, Java, Big Data, AWS]

Removing items from HashSet using remove()

Iterating over course list after removing Python:

Java

JavaScript

Big Data

Node.js

AWS

Machine Learning

Courses Hashset :[Java, JavaScript, Big Data, Node.js, AWS, Machine Learning]

New Courses Hashset :[Node.js, Machine Learning, Python]

Remove NewCourses hashset from main courses hashset

After invoking removeAll() method courses left: [Java, JavaScript, Big Data, AWS]

After invoking removeIf(Java) method: [Big Data, AWS]

Add Python and Node.js again in the courses

Updated Courses hashset:[Big Data, Node.js, AWS, Python]

HashSet after retainAll() operation : [Node.js, Python]

After invoking clear() method: []

### **How HashSet add method works**

Why it returns false in-case of duplicate element

public class HashSet extends AbstractSet implements Set, Cloneable, java.io.Serializable

{

**private transient HashMap<E,Object> map;**

// PRESENT is dummy value which will be used as value in map

private static final Object PRESENT = new Object();

/\*\*Constructs an empty map. So hash \*/

public HashSet() {

map = new HashMap<E,Object>();

}

// return false if e is already present in HashSet

public boolean add(E e) {

return map.put(e, PRESENT)==null; // if element is not present in the map then it returns true

}

// other HashSet methods

}

HashSet in Java uses HashMap to store the object. One HashMap object is created when a HashSet object is created. The elements or objects entered in the HashSet are stored as keys in the HashMap.

**How will you remove duplicate elements from an ArrayList?**

To remove duplicate elements from an ArrayList, Use Set.

**Option 1:**

Use Set if ordering of elements is not important. We will just put the elements of ArrayList in a HashSet and then add them back to the ArrayList.

List<Integer> list = new ArrayList<Integer>();

list.add(5);

list.add(10);

Set<Integer> s = new HashSet<Integer>(list);

list.clear();

list.addAll(s);

**Option 2:**

Use LinkedHashSet if ordering of elements is important. Then we will put the elements of ArrayList in a LinkedHashSet and then add them back to the ArrayList.

List<Integer> list = new ArrayList<Integer>();

list.add(5);

list.add(15);

Set<Integer> s = new LinkedHashSet<Integer>(list);

list.clear();

list.addAll(s);

## **LinkedHashSet**

* Child of HashSet
* Same as HashSet – No duplicate allowed , Insertion Order preserved

Why – Because underline data structure is Hashtable + Linked List

Main feature of the LinkedHashSet

* It extends HashSet class which implements Set interface.
* Duplicate values are not allowed in LinkedHashSet.
* One NULL element is allowed in LinkedHashSet.
* It is an **ordered collection** in which elements were inserted into the set and it maintain the order (insertion-order).
* Like HashSet, this class offers constant time performance for the basic operations(add, remove, contains and size).
* LinkedHashSet is not synchronized. If multiple threads access a hash set concurrently, and at least one of the threads modifies the set, it must be synchronized externally.
* Use Collections.synchronizedSet(new LinkedHashSet()) method to get the synchronized LinkedHashSet.
* The iterators returned by this class’s iterator method are fail-fast and may throw ConcurrentModificationException if the set is modified at any time after the iterator is created, in any way except through the iterator’s own remove() method.
* LinkedHashSet also implements Searlizable and Cloneable interfaces.

HashSet

1. Underline data structure is HashMap
2. Insertion Order not persevered
3. Came in Java 1.2
4. Better performance than LinkedHashSet

LinkedHashSet

1. Underline data structure is HashTable + Linked List
2. Insertion Order persevered
3. Came in java 1.4
4. Performance is likely to be just slightly below that of HashSet, due to the added expense of maintaining the linked list

## **TreeSet**

* Underline data structure is balance tree.
* Duplicates not allowed
* Insertion order not maintained
* Elements are inserted with sorting
* Heterogenous objects are not allowed – (In treemap also as both are sorted)
* You will get :ClassCastException runtime exception
* Null not allowed (will give NullPointerException as if we add null then it will try to compare it)

Constructor

* TreeSet t= new TreeSet() default natural sorting order.
* TreeSet t=new TreeSet(Comparator c) – Custom sorting using Comparator
* TreeSet t=new TreeSet(Collection c)
* TreeSet t=new TreeSet(SortedSet s)

### **Code Example TreeSet**

package com.learning.corejava.basics;

import java.util.Iterator;

import java.util.TreeSet;

public class TreeSetExample {

public static void main(String args[]) {

TreeSet<Integer> evenNumbers = new TreeSet<>();

// Using the add() method. It add the elements with sorting.

evenNumbers.add(2);

evenNumbers.add(4);

evenNumbers.add(66);

evenNumbers.add(6);

evenNumbers.add(14);

evenNumbers.add(10);

evenNumbers.add(null);

System.out.println("TreeSet: " + evenNumbers);

// Using the remove() method

boolean value1 = evenNumbers.remove(10);

System.out.println("Is 10 removed? " + value1);

System.out.println("TreeSet after removal of 10: " + evenNumbers);

// Using the first() method. Only available in TreeSet.

int first = evenNumbers.first();

System.out.println("First Number: " + first);

// Using the last() method

int last = evenNumbers.last();

System.out.println("Last Number: " + last);

// Using higher() : Returns the element that are greater than the specified element.

System.out.println("Using higher: " + evenNumbers.higher(14));

// Using lower() Returns the element that are less than the specified element.

System.out.println("Using lower: " + evenNumbers.lower(10));

// Using ceiling() Returns the elements that is greater than the specified element. If the element passed exists in a tree set, it returns the element passed as an argument

System.out.println("Using ceiling: " + evenNumbers.ceiling(14));

// Using floor()

System.out.println("Using floor: " + evenNumbers.floor(9));

// Calling iterator() method

Iterator<Integer> iterate = evenNumbers.iterator();

System.out.print("TreeSet using Iterator: ");

// Accessing elements

while(iterate.hasNext()) {

System.out.print(iterate.next());

System.out.print(", ");

}

}

}

### **Null inside TreeSet**

* From 1.7 onwards null is not at all accepted by TreeSet. If you enforce to add then we will get NullPointerException. Till 1.6 null was accepted only as the first element.
* NullPointerException - if the specified element is null and this set uses natural ordering, or its comparator does not permit null elements
* So since you have not specified a custom comparator implementation that can handle null values, you get the NPE.

It was possible to add a null element as the first element of a TreeSet/TreeMap in Java 6, but it was considered a bug:

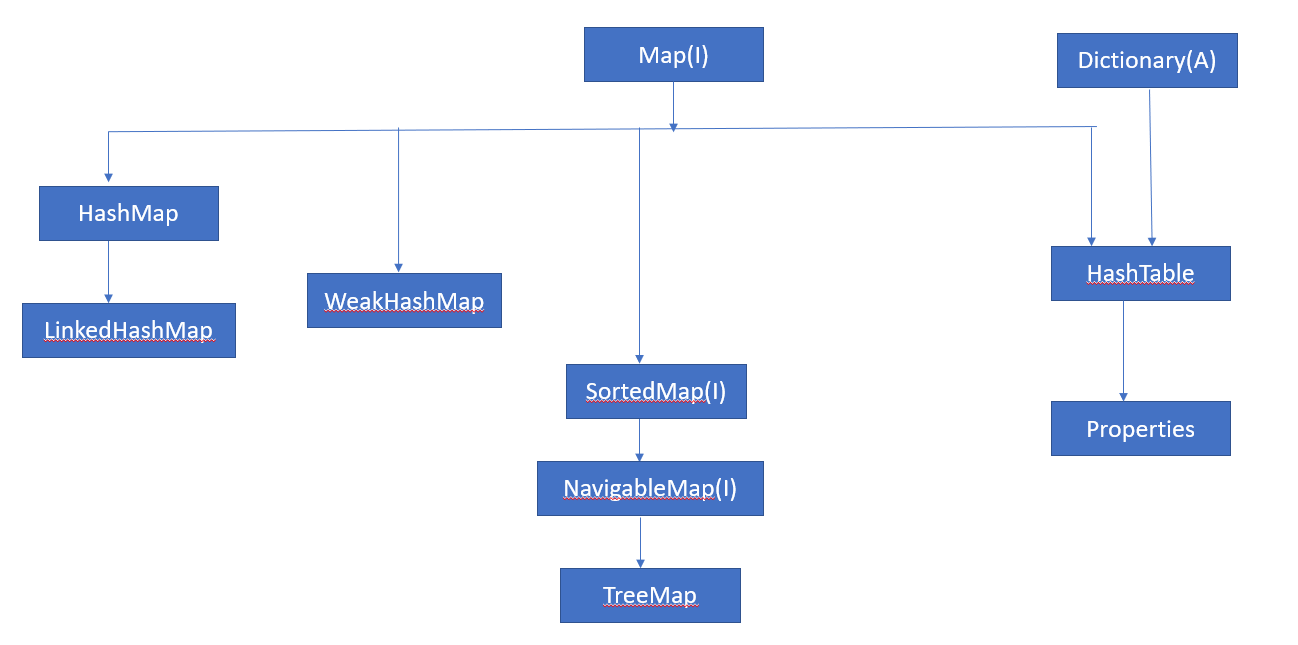
## Difference: HashSet, LinkedHashSet, TreeSet

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **HashSet** | **LinkedHashSet** | **TreeSet** |
|  |  |  |  |
|  |  |  |  |
| Underline DS | HashMap | HashTable + LinkedList | Balanced Tree |
| Insertion Order | Not Preserved – Element inserted according to hashcode | Preserved as inserted | Not applicable |
| Sorting Order | Not applicable | Not applicable | Sorted as per natural order |
| Heterogeneous Object | Allowed | Allowed | Not allowed. Gives ClassCastException |
| Duplicate objects | Not allowed | Not allowed | Not allowed |
| Null acceptance | Allowed. Only once | Allowed Only once | Not allowed. As TreeSet needs to check the comparison. |
|  |  |  |  |

# Difference: List and Set

|  |  |
| --- | --- |
| The list maintains insertion order. | Set do not maintain any insertion order. |
| The list provides index access of the elements in the collection. | The set does not provides index access of the elements in the collection. |
| The list can store multiple null elements | Set can store only one null element. |
| The list interface allows duplicate elements | The set interface don’t allows duplicate elements |
| Use List if you wanaa access the elements frequently (get() method) | If you wanna create a collection of unique elements then use set |
| Implementaion : ArrayList, LinkedList , Vector | Implementation: HashSet, TreeSet, LinkedHashSet |

# Java Map Interface



## Map Interface

### **What is Map**

*Designed for the faster lookups.*

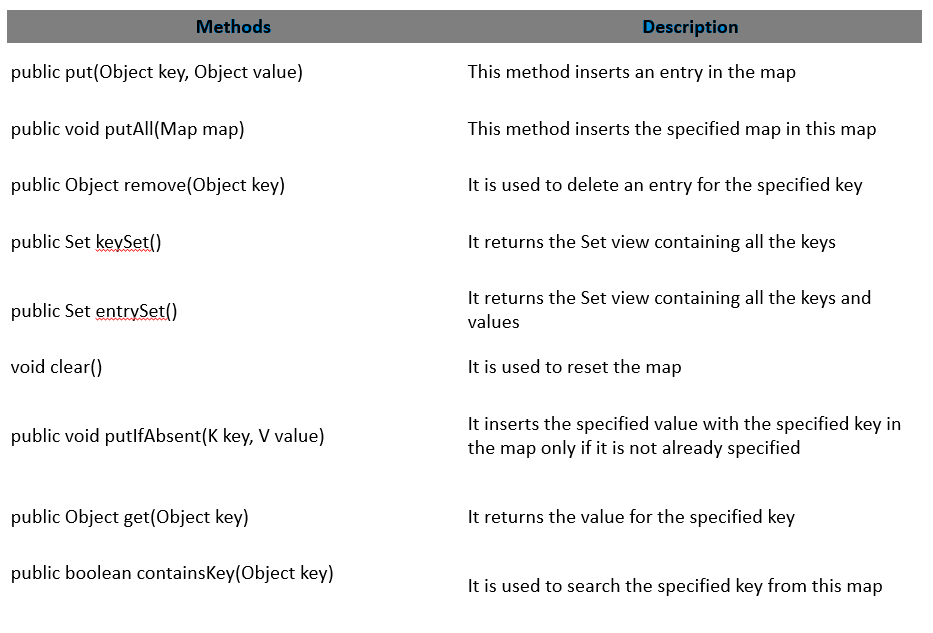
*Data is stored in key-value pairs and every key is unique*

*Each key maps to a value hence the name map.*

*These key-value pairs are called map entries.*

*-* A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null value *(HashMap and LinkedHashMap)* but some does not *(TreeMap).*

### **Map Methods**



Question.

What does map put() method return.

Answer:

Sop(map.put(101,"Kunal"))

# will return null as no key is present earlier in the map. New Key-value pair will be added and put method will return null.

Sop(map.put(101,"Kumar")) #key is already present. Thus it return old value as "Kunal" and new value Kumar will get updated in the map.

## **HashMap**

Java HashMap is a Key-Value Pair implementation that implements Map interface and It works similar to Hash Table.

* As I mentioned above, Java HashMap is a Key-Value pair Java Map interface, which is used to store the Objects as values and those objects should have Keys to identify them.
* Java HashMap cannot have duplicate keys but it can have duplicates values.
* Hash Map can have only one NULL key and multiple NULL Values.
* The Order in which, We add the values to HashMap is not guaranteed, hence the Insertion order is not Preserved.
* Searching the Object is fast since the Java HashMap has the Keys.
* HashMap is not good for Multi-Threading because it does not support Synchronization.
* HashMap keys are immutable (only immutable keys are allowed)

Performance of HashMap depends on 2 parameters:

* Initial Capacity
* Load Factor
* **Initial Capacity** – It is the capacity of HashMap at the time of its creation (It is the number of buckets a HashMap can hold when the HashMap is instantiated). In java, it is 2^4=16 initially, meaning it can hold 16 key-value pairs.
* **Load Factor** – It is the percent value of the capacity after which the capacity of Hashmap is to be increased (It is the percentage fill of buckets after which Rehashing takes place). In java, it is 0.75f by default, meaning the rehashing takes place after filling 75% of the capacity.

**HashMap Example 1**

package com.learning.corejava.collection;

import java.util.Collection;

import java.util.HashMap;

import java.util.Map;

import java.util.Map.Entry;

import java.util.Set;

public class HashMapBasicExample {

// HashMap(non-synchronize - don't use it in thread.) Implementation of Map interface.

// Other Implementations are: HashTable, TreeMap, ConcurrentHashMap (use it in threads)

public static void main(String[] args) {

// Key must be unique & it must be immutable(String or simple literal or Object).

Map<String, String> map = new HashMap<>();

map.put("1", "1"); // put example

map.put("2", "2");

map.put("3", "3");

map.put("4", null); // null value allowed.

map.put(null, "100"); // One null key

Map mTemp=new HashMap<>();

mTemp.put(1, "ONE");

mTemp.put(1.5, "ONE Point Five");

System.out.println("Temp map"+mTemp);

String value = map.get("3"); // get example with time complaxity.(O(1))

System.out.println("Key = 3, Value = " + value);

value = map.getOrDefault("5", "Default Value");

System.out.println("Key = 5, Value=" + value);

//Add duplicate key

map.put("1", "11");

System.out.println("Map after duplicate key(1) added:"+map);

boolean keyExists = map.containsKey(null);

boolean valueExists = map.containsValue("100");

System.out.println("keyExists=" + keyExists + ", valueExists=" + valueExists);

Set<Entry<String, String>> entrySet = map.entrySet();

System.out.println("\n"+entrySet);

System.out.println("map size=" + map.size());

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

map1.putAll(map);

System.out.println("map1 mappings= " + map1);

String nullKeyValue = map1.remove(null);

System.out.println("map1 null key value = " + nullKeyValue);

System.out.println("map1 after removing null key = " + map1);

Set<String> keySet = map.keySet();

System.out.println("map keys = " + keySet);

Collection<String> values = map.values();

System.out.println("map values = " + values);

map.clear();

System.out.println("map is empty=" + map.isEmpty());

}

}

**HashMap Example 2**

**package** com.learning.corejava.collection;

**public** **class** Product {

**private** String item;

**private** **int** price;

**public** Product(String it, **int** pr){

**this**.item = it;

**this**.price = pr;

}

/\* equal and hash-code contract

\* 1) If two objects are equal(by equal method) then they must produce same hash-code.

\* 2) Different objects can have same hash-code.(collision).

\*

\*/

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

System.***out***.println("In hashcode");

**int** hashcode = 0;

hashcode = (price\*16)/2;

hashcode += item.hashCode();

**return** hashcode;

}

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

System.***out***.println("In equals");

**if** (obj **instanceof** Product) {

Product pp = (Product) obj;

**return** (pp.item.equals(**this**.item) && pp.price == **this**.price);

} **else** {

**return** **false**;

}

}

**public** String getItem() {

**return** item;

}

**public** **void** setItem(String item) {

**this**.item = item;

}

**public** **int** getPrice() {

**return** price;

}

**public** **void** setPrice(**int** price) {

**this**.price = price;

}

**public** String toString(){

**return** "(Product item name: "+item+",price: "+price+")";

}

}

**package** com.learning.corejava.collection;

**import** java.util.HashMap;

**public** **class** HashMapProductDemo {

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

HashMap<Product,String> hm=**new** HashMap<>();

hm.put(**new** Product("Apple",100),"Apple");

hm.put(**new** Product("Orange",80),"Orange");

hm.put(**new** Product("Banana",50),"Banana");

Product temp=**new** Product("Pinaple",80);

hm.put(temp,"Pinaple"); // Deep Copy.

// Below will change the value of Apple product from Apple to Apple1

//hm.put(new Product("Apple",100),"Apple1");

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

temp.setPrice(100);

Product p=**new** Product("Orange",80);

System.***out***.println(hm.containsKey(p));

System.***out***.println("END");

}

}

HashMap interview questions

Why is null allowed in HashMap?

HashMap is newer than Hashtable and fixes some of its limitations.

* *For HashMap, it allows one null key and there is a null check for keys, if the key is null then that element will be stored in a zero location in Entry array.*
* We cannot have more than one Null key in HashMap because **Keys are unique** therefor only one Null key and many Null values are allowed.

Null not allowed in hashtable.

To successfully store and retrieve objects from a HashTable, the objects used as keys must implement the hashCode method and the equals method. Since null is not an object, it can’t implement these methods.

How HashMap works internally?

HashMap internal code

/\* transient HashMapEntry<K,V>[] table

\*

\* class HashMapEntry<K,V> {

final K key;

V value;

HashMapEntry<K,V> next;

int hash;

// Some utility methods

}

\*

\*

\*/

HashMap works on the principle of hashing, we have *put(key, value)* and *get(key)* method for storing and retrieving Objects from HashMap.

When we pass Key and Value object to put() method on Java HashMap, HashMap implementation calls [hashCode method](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html)on Key object and applies returned hashcode into its own hashing function to find a bucket location for storing Entry object, the important point to mention is that [HashMap in Java](https://www.java67.com/2013/02/10-examples-of-hashmap-in-java-programming-tutorial.html) stores both key and value object as Map.Entry in a bucket is essential to understand the retrieving logic.

Now the question comes for collision.

[*equals() and hashCode()* contract](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) that two unequal objects in Java can have the same hashcode.  
  
"Since hashcode is same, bucket location would be same and collision will occur in HashMap Since HashMap uses LinkedList to store object, this entry (object of Map.Entry comprise key and value )  will be stored in [LinkedList](http://javarevisited.blogspot.sg/2012/02/difference-between-linkedlist-vs.html).

# **Concurrency**

* Two task can operate concurrently and both make a progress
* Before multi-core processor

OS rapidly switch between different task, doing small portion of each task before moving to next so that all can be progress.

## **Multithread**

* Java makes concurrency available to the programmer using java.lang.Thread API’s
* Java programs can have multiple threads of execution
* Each thread has its own method stack and PC, allow it to use concurrency with other threads.

If Exception occurs, then only that thread will get terminated.

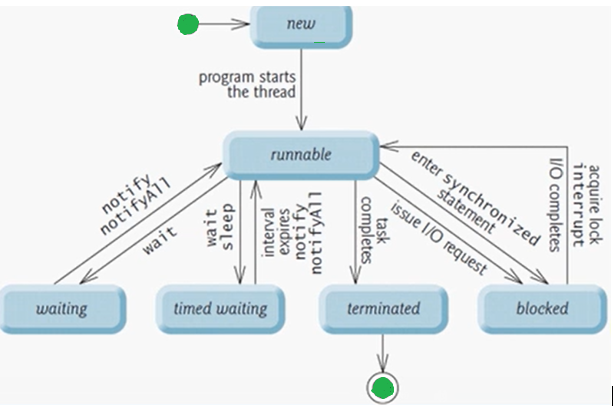
## Example of concurrency

* Stream video on the internet
  + Producer thread – download the video
  + Consumer thread – play the video

These activities perform simultaneously

* Threads are synchronized – coordinated.
  + To avoid choppy playback, the player thread does not being until there’s a sufficient amount of video available in memory.

## **Life cycle of a thread**



Three ways to create thread  
There are three ways you can specify your code to be executed by a thread

1. By inheriting your class from the Thread class
2. By implementing the Runnable interface in your class
3. By using the [method reference](http://java-latte.blogspot.in/2014/03/method-references-in-java-8.html) to a method that takes no parameters and returns void

### **By inheriting your class from the Thread class**

When you inherit your class from the Thread class, you should override the run() method and provide the code to be executed by the thread.

**package** com.learning.threads;

**class** MyThread **extends** Thread{

@Override

**public** **void** run() {

System.***out***.println("Running thread..");

}

}

**public** **class** InheritsThreadClassDemo {

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

MyThread t=**new** MyThread();

t.start();

}

}

### **By implementing the Runnable interface in your class**

You can create a class that implements the java.lang.Runnable interface. Runnable is a [functional interface](http://java-latte.blogspot.in/2014/02/functional-interface-and-lambda-in-java.html) and it is declared as follows  
  
@[FunctionalInterface](http://java-latte.blogspot.in/2014/02/functional-interface-and-lambda-in-java.html)  
**public interface Runnable**

**package** com.learning.threads;

**import** java.util.concurrent.TimeUnit;

**public** **class** RunnableInterfaceDemo {

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

Runnable runDemo = () -> System.***out***.println("Thread Running");

Thread t=**new** Thread(runDemo);

t.start();

Runnable runnable = () -> {

**try** {

String name = Thread.*currentThread*().getName();

System.***out***.println("Foo " + name);

TimeUnit.***SECONDS***.sleep(1);

System.***out***.println("Bar " + name);

}

**catch** (InterruptedException e) {

e.printStackTrace();

}

};

Thread thread = **new** Thread(runnable);

thread.start();

}

}

Output :

Thread Running

Foo Thread-1

Bar Thread-1

When you run the above code you'll notice the one second delay between the first and the second print statement. TimeUnit is a useful enum for working with units of time. Alternatively you can achieve the same by calling Thread.sleep(1000).

### **By using the**[**method reference**](http://java-latte.blogspot.in/2014/03/method-references-in-java-8.html)**to a method that takes no parameters and returns void**

**package** com.learning.threads;

**class** ThreadTest{

**public** **static** **void** execute() {

System.***out***.println("Thread using method reference");

}

}

**public** **class** ThreadUsingMethodReferenceDemo {

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

Thread t=**new** Thread(ThreadTest :: *execute*);

t.start();

}

}

If you don’t understand method reference example then Please see below example.

Java 8 lambda expressions avoids creating objects for functional interface. If in the application if there is already some method in some class, which we feel, is a perfect implementation of functional method of functional interface, wouldn’t be nice if we could refer this existing methods instead of using a lambda expression? This is exactly what we can do using method references.



## **Thread synchronization**

When multiple thread shares an object and its modified by one or more thread, Indeterminate state may occur.

Give only one thread an exclusive access. (Mutual exclusion)

Java built in Monitors

Every Object has a monitor and a monitor lock

The monitor ensure that its objects monitor lock is held by max one thread at any time.

It is used to enforce the mutual exclusion.

To specify that a thread must hold the monitor lock to execute a block of code, the code should be placed in synchronized statement.

synchronized(object) { }

Example to demonstrate how unsynchronized method is not thread safe.

Example will create a two thread that will write the integer data to shared array. One thread will try to write 1,2,3(start with 1) and second will try to write 11,12,13(Thread initialize to start at 11). Eventually one may override other’s data.

**package** com.learning.threads;

**import** java.security.SecureRandom;

**import** java.util.Arrays;

// class that will write the data into an array.

**public** **class** SimpleArray {

**private** **final** **static** SecureRandom ***sRandom***= **new** SecureRandom();

**private** **int** arr[]; //shared array

**private** **int** writeInx = 0; //shared index for next int to writeInx

**public** SimpleArray(**int** size){

arr = **new** **int**[size];

}

**public** **void** add(**int** value){

**int** position= writeInx;

**try** {

Thread.*sleep*(***sRandom***.nextInt(500));

}

// This is checked exception and must be handle.

**catch** (InterruptedException e) {

e.printStackTrace();

}

arr[position]=value;

System.***out***.printf("%s wrote %2d to element %d \n",Thread.*currentThread*().getName(),value,position);

// May be it will happen that one thread will add value to array and then other thread will override that value.

++writeInx; // increment index of element to be written next time.

System.***out***.printf("Next write index : %d \n",writeInx);

}

**public** String toString() {

**return** Arrays.*toString*(arr);

}

}

**package** com.learning.threads;

// Call add method that will Add integer to an array shared with other Runnable threads.

**public** **class** ArrayWriter **implements** Runnable{

**private** **final** SimpleArray sharedSimpleArray;

**private** **final** **int** startValue;

**public** ArrayWriter(**int** value,SimpleArray array) {

**this**.startValue = value;

**this**.sharedSimpleArray = array;

}

**public** **void** run() {

**for** (**int** i=startValue;i<startValue+3;i++) {

sharedSimpleArray.add(i); // add element to shared array

}

}

}

// Test class

**package** com.learning.threads;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.TimeUnit;

**public** **class** SharedArrayTest {

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

// Integer array will have 6 element.

SimpleArray sharedSimpleArray = **new** SimpleArray(6);

// Create two task to write to the shared SimpleArray

// First writer will start writing from 1 and second with 11

ArrayWriter writer1= **new** ArrayWriter(1, sharedSimpleArray);

ArrayWriter writer2= **new** ArrayWriter(11, sharedSimpleArray);

//execute the task with an ExecutorService

ExecutorService executorService = Executors.*newCachedThreadPool*();

executorService.execute(writer1);

executorService.execute(writer2);

executorService.shutdown();

**try** {

// wait the termination of task to finish

**boolean** taskend = executorService.awaitTermination(1, TimeUnit.***MINUTES***);

**if**(taskend) {

System.***out***.printf("%n Content of SimpleArray:%n");

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

}

**else**

System.***out***.println("Timed out while waiting of the task finished.");

}

**catch**(InterruptedException e) {

e.printStackTrace();

}

}

}

Output 1:

pool-1-thread-1 wrote 1 to element 0

Next write index : 1

pool-1-thread-2 wrote 11 to element 0

Next write index : 2

pool-1-thread-1 wrote 2 to element 1

Next write index : 3

pool-1-thread-2 wrote 12 to element 2

Next write index : 4

pool-1-thread-2 wrote 13 to element 4

Next write index : 5

pool-1-thread-1 wrote 3 to element 3

Next write index : 6

Content of SimpleArray:

[11, 2, 12, 3, 13, 0]

Output 2:

pool-1-thread-2 wrote 11 to element 0

Next write index : 1

pool-1-thread-1 wrote 1 to element 0

Next write index : 2

pool-1-thread-2 wrote 12 to element 1

Next write index : 3

pool-1-thread-1 wrote 2 to element 2

Next write index : 4

pool-1-thread-2 wrote 13 to element 3

Next write index : 5

pool-1-thread-1 wrote 3 to element 4

Next write index : 6

Content of SimpleArray:

[1, 12, 2, 13, 3, 0]

Solution of unthread safe add method

**public** **synchronized** **void** add(**int** value){

--------------

}

In our SharedArrayTest main method after we create our ArrayWriters and we launch our tasks, the tasks start executing. In the ArrayWriter class, when add() method executes, before the actual method call is made, first, a determination is made as to whether or not the monitor lock for the shared simple array object is available. If it is available, then this call to add proceeds, and during the time that the body of this method is executing, the add method or the thread that calls the add method, rather, holds onto the SimpleArray's monitor lock.

Only when the end of this method is reached, the monitor lock released, so while one thread is executing in the body of add method, if a second thread comes along and tries to perform this add call, that second thread is automatically blocked by the operating system from continuing to execute until the lock becomes available, and then only at that point, if the thread gets scheduled to execute, will it then obtain the lock.

Output:

pool-1-thread-1 wrote 1 to element 0

Next write index : 1

pool-1-thread-1 wrote 2 to element 1

Next write index : 2

pool-1-thread-1 wrote 3 to element 2

Next write index : 3

pool-1-thread-2 wrote 11 to element 3

Next write index : 4

pool-1-thread-2 wrote 12 to element 4

Next write index : 5

pool-1-thread-2 wrote 13 to element 5

Next write index : 6

Content of SimpleArray:

[1, 2, 3, 11, 12, 13]

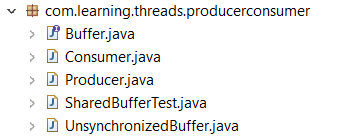
## Producer consumer problem

**Producer-consumer without synchronization**

In this example producer will write int values to shared buffer and consumer consumes it. At the end the sum of values produce and consume should be same.

**Note: %n and \n difference**

There is also one specifier that doesn't correspond to an argument. It is "%n" which outputs a line break. A "\n" can also be used in some cases, but since "%n" always outputs the correct platform-specific line separator, it is portable across platforms whereas"\n" is not



**// buffer.java**

**package** com.learning.threads.producerconsumer;

// Buffer interface specifies methods called by producer and consumer

**public** **interface** Buffer {

//place int value to buffer

**public** **void** blockingPut(**int** value) **throws** InterruptedException;

// Return end value from buffer

**public** **int** blockingGet() **throws** InterruptedException;

}

**// Buffer implementation: UnsynchronizedBuffer.java**

**package** com.learning.threads.producerconsumer;

**public** **class** UnsynchronizedBuffer **implements** Buffer {

**private** **int** buffer=-1; // shared by producer and consumer thread

@Override

**public** **void** blockingPut(**int** value) **throws** InterruptedException {

System.***out***.printf("Producer writes\t%2d",value);

buffer = value;

}

@Override

**public** **int** blockingGet() **throws** InterruptedException {

System.***out***.printf("Consumer reads\t%2d",buffer);

**return** buffer;

}

}

**// Producer thread. Producer.java**

**package** com.learning.threads.producerconsumer;

**import** java.security.SecureRandom;

**public** **class** Producer **implements** Runnable {

**private** **static** **final** SecureRandom ***generator***= **new** SecureRandom();

**private** **final** Buffer sharedBufferLocation; // reference to shared object

**public** Producer(Buffer sharedLocation) {

**this**.sharedBufferLocation = sharedLocation;

}

// Store values from 1 to 10 in sharedLocation

@Override

**public** **void** run() {

**int** sum=0;

**for**(**int** count=1;count<=10;count++) {

**try** {

Thread.*sleep*(***generator***.nextInt(3000));// Random sleep

sharedBufferLocation.blockingPut(count); // Set value in buffer

sum += count;

System.***out***.printf("\t%2d%n",sum);

}**catch**(InterruptedException e) {

Thread.*currentThread*().interrupt();

}

}

System.***out***.printf("Producer done producing%nTerminating producer%n");

}

}

**// Consumer.java**

**package** com.learning.threads.producerconsumer;

**import** java.security.SecureRandom;

**public** **class** Consumer **implements** Runnable {

**private** **static** **final** SecureRandom ***generator***= **new** SecureRandom();

**private** **final** Buffer sharedBufferLocation; // reference to shared object

**public** Consumer(Buffer sharedLocation) {

**this**.sharedBufferLocation = sharedLocation;

}

// Store values from 1 to 10 in sharedLocation

@Override

**public** **void** run() {

**int** sum=0;

**for**(**int** count=1;count<=10;count++) {

**try** {

// sleep 0 to 3 sec, read value from buffer and add to the sum

Thread.*sleep*(***generator***.nextInt(3000));// Random sleep

sum += sharedBufferLocation.blockingGet(); // Set value in buffer

System.***out***.printf("\t\t\t%2d%n",sum);

}**catch**(InterruptedException e) {

Thread.*currentThread*().interrupt();

}

}

System.***out***.printf("%n%s %d%n%s%n",

"Consumer read values total=",sum,"Terminating consumer");

}

}

**// Test class**

package com.learning.threads.producerconsumer;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

public class SharedBufferTest {

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

// Create new thread pool with two threads.

ExecutorService executorsService= Executors.newCachedThreadPool();

Buffer sharedBufferLocation=new UnsynchronizedBuffer();

System.out.printf(

"Action\t\tValue\tSum of Produced\tSum of consumed%n");

System.out.printf(

"------\t\t-----\t---------------\t---------------%n%n");

// Execute producer and consumer giving each the access to sharedBufferLocation

executorsService.execute(new Producer(sharedBufferLocation));

executorsService.execute(new Consumer(sharedBufferLocation));

executorsService.shutdown();

executorsService.awaitTermination(2, TimeUnit.MINUTES);

}

}

Output // As producer and consumer not in sync, they run as they schedule and might get into unexpected result. Expected consumer sum is 55

// First run

Action Value Sum of Produced Sum of consumed

------ ----- --------------- ---------------

Consumer reads -1 -1

Producer writes 1 1

Consumer reads 1 0

Producer writes 2 3

Consumer reads 2 2

Producer writes 3 6

Producer writes 4 10

Consumer reads 4 6

Producer writes 5 15

Consumer reads 5 11

Producer writes 6 21

Producer writes 7 28

Consumer reads 7 18

Consumer reads 7 25

Producer writes 8 36

Producer writes 9 45

Consumer reads 9 34

Consumer reads 9 43

Consumer reads 9 52

Consumer read values total= 52

Terminating consumer

Producer writes 10 55

Producer done producing

Terminating producer

**// Second run**

Action Value Sum of Produced Sum of consumed

------ ----- --------------- ---------------

Producer writes 1 1

Consumer reads 1 1

Consumer reads 1 2

Producer writes 2 3

Consumer reads 2 4

Consumer reads 2 6

Producer writes 3 6

Producer writes 4 10

Consumer reads 4 10

Producer writes 5 15

Consumer reads 5 15

Producer writes 6 21

Consumer reads 6 21

Consumer reads 6 27

Producer writes 7 28

Producer writes 8 36

Producer writes 9 45

Consumer reads 9 36

Consumer reads 9 45

Consumer read values total= 45

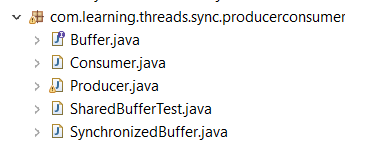
Terminating consumer

Producer writes 10 55

Producer done producing

Terminating producer

## Solution to producer consumer problem.



Changes

**package** com.learning.threads.sync.producerconsumer;

**public** **class** SynchronizedBuffer **implements** Buffer {

**private** **int** buffer=-1; // shared by producer and consumer thread

**private** **boolean** occupied=**false**;

@Override

// until monitor lock gets released which is held by producer consumer can't execute get method

**public** **synchronized** **void** blockingPut(**int** value) **throws** InterruptedException {

//while there are not empty locations - place thread in waiting

**while**(occupied) {

System.***out***.println("Producer tried to writes");

displayState("Buffer full. Producer waits");

wait();

}

buffer = value;

//indicate producer can't store another value until consumer read current one

occupied=**true**;

displayState("Producer writes "+buffer);

notifyAll(); //tell waiting threads to enter runnable states

}// end method blockingPut, releases the lock on synchronizedBuffer.

**private** **synchronized** **void** displayState(String operation) {

System.***out***.printf("%-40s%d\t\t%b%n%n",operation,buffer,occupied);

}

@Override

// consumer cant get into this method if monitor lock is occupied by producer

**public** **synchronized** **int** blockingGet() **throws** InterruptedException {

// while no data to read place the thread in sleep state

**while**(!occupied) {

System.***out***.println("Consumer tried to reads");

displayState("Buffer empty. Consumer waits");

wait();

}

// indicate that producer can run and store another value

occupied=**false**;

displayState("Consumer reads:"+buffer);

notifyAll();

**return** buffer;

}//end of blocking Get, releases the lock on synchronizedBuffer.

}

## Fork-Join

The fork-join framework allows to break a certain task on several workers and then wait for the result to combine them. It leverages multi-processor machine's capacity to great extent.

**Fork**

Fork is a process in which a task splits itself into smaller and independent sub-tasks which can be executed concurrently

Syntax

Sum left = new Sum(array, low, mid);

left.fork();

Here Sum is a subclass of RecursiveTask and left.fork() spilts the task into sub-tasks.

**Join**

Join is a process in which a task join all the results of sub-tasks once the subtasks have finished executing, otherwise it keeps waiting.

Syntax

left.join();

Here left is an object of Sum class

**ForkJoinPool**

it is a special thread pool designed to work with fork-and-join task splitting.

Syntax

ForkJoinPool forkJoinPool = new ForkJoinPool(4);

Here a new ForkJoinPool with a parallelism level of 4 CPUs.

**RecursiveTask**

RecursiveTask represents a task which returns a value.

Syntax

class Sum extends RecursiveTask<Long> {

@Override

protected Long compute() { return null; }

}

**Example ForkJoin**

**package** com.learning.threads;

**import** java.util.concurrent.ExecutionException;

**import** java.util.concurrent.ForkJoinPool;

**import** java.util.concurrent.RecursiveTask;

**public** **class** TestForkJoin {

**public** **static** **void** main(**final** String[] arguments) **throws** InterruptedException,

ExecutionException {

**int** nThreads = Runtime.*getRuntime*().availableProcessors();

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

**int**[] numbers = **new** **int**[10000];

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

numbers[i] = i;

}

/\* Compute the sum of 20 for test purpose.

\* int sum=0;

for(int i = 0; i < 20; ++i)

sum += numbers[i];

System.out.println("Sum="+sum);\*/

ForkJoinPool forkJoinPool = **new** ForkJoinPool(nThreads);

Long result = forkJoinPool.invoke(**new** Sum(numbers,0,numbers.length));

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

}

**static** **class** Sum **extends** RecursiveTask<Long> {

**int** low;

**int** high;

**int**[] array;

Sum(**int**[] array, **int** low, **int** high) {

**this**.array = array;

**this**.low = low;

**this**.high = high;

}

**protected** Long compute() {

**if**(high - low <= 10) {

**long** sum = 0;

**for**(**int** i = low; i < high; ++i)

sum += array[i];

**return** sum;

} **else** {

**int** mid = low + (high - low) / 2;

Sum left = **new** Sum(array, low, mid);

Sum right = **new** Sum(array, mid, high);

left.fork();

**long** rightResult = right.compute();

**long** leftResult = left.join();

**return** leftResult + rightResult;

}

}

}

}

**Interview Questions**

**1) Which collection classes are thread-safe in Java?**

A **thread-safe** class is a class that guarantees the internal state of the class as well as returned values from methods, are correct while invoked concurrently from multiple threads. The collection classes that are **thread-safe** in Java are **Stack**, **Vector**, **Properties**, **Hashtable**, etc.

Stack

The **Stack**class in Java implements the stack data structure that is based on the principle of **LIFO**. So, the **Stack**class can support many operations such as **push, pop, peek, search, empty**, etc.

Example

import java.util.\*;

public class StackTest {

   public static void main (String[] args) {

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

      stack.push(5);

      stack.push(7);

      stack.push(9);

      Integer num1 = (Integer)stack.pop();

      System.out.println("The element popped is: " + num1);

      Integer num2 = (Integer)stack.peek();

      System.out.println(" The element on stack top is: " + num2);

   }

}

Output

The element popped is: 9

The element on stack top is: 7

**2) String is thread safe in Java**

**String** is immutable ( once created cannot be changed )object . The object created as a **String** is stored in the Constant **String** Pool. Every immutable object in **Java** is **thread safe** , that implies **String** is also **thread safe** . **String** cannot be used by two **threads** simultaneously.

**StringBuffer**

StringBuffer is mutable means one can change the value of the object . The object created through StringBuffer is stored in the heap. StringBuffer has the same methods as the StringBuilder , but each method in StringBuffer is synchronized that is StringBuffer is thread safe .

Due to this it does not allow two threads to simultaneously access the same method . Each method can be accessed by one thread at a time .

But being thread safe has disadvantages too as the performance of the StringBuffer hits due to thread safe property .

Thus StringBuilder is faster than the StringBuffer when calling the same methods of each class.

String Buffer can be converted to the string by using toString() method.

StringBuffer demo1 = new StringBuffer("Hello") ; // object stored in heap and its value can be changed .

demo1=new StringBuffer("Bye"); // statement is right as it modifies the value which is allowed in the StringBuffer

**StringBuilder**

StringBuilder is same as the StringBuffer , that is it stores the object in heap and it can also be modified . The main difference between the StringBuffer and StringBuilder is that StringBuilder is also not thread safe. StringBuilder is fast as it is not thread safe .

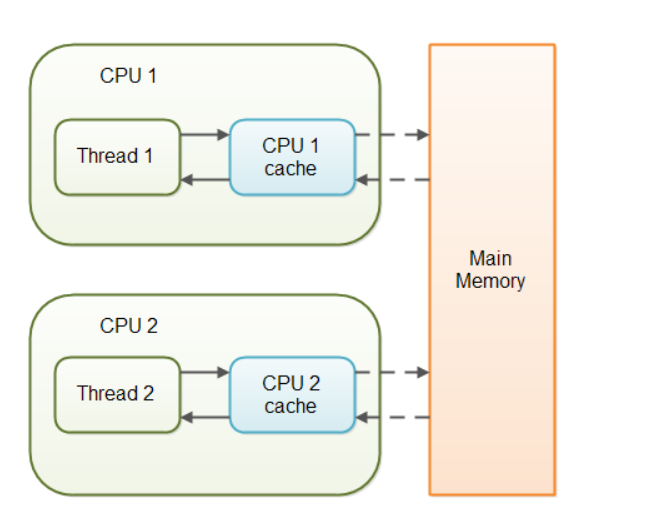
StringBuilder demo2= new StringBuilder("Hello"); // object too is stored in the heap and its value can be modified

demo2=new StringBuilder("Bye"); // statement is right as it modifies the value which is allowed in the StringBuilder.

**3) What is volatile variable.**

The Java volatile keyword guarantees visibility of changes to variables across threads. This may sound a bit abstract, so let me elaborate.

In a multithreaded application where the threads operate on non-volatile variables, each thread may copy variables from main memory into a CPU cache while working on them, for performance reasons. If your computer contains more than one CPU, each thread may run on a different CPU. That means, that each thread may copy the variables into the CPU cache of different CPUs.



With non-volatile variables there are no guarantees about when the Java Virtual Machine (JVM) reads data from main memory into CPU caches, or writes data from CPU caches to main memory. This can cause several problems which I will explain in the following sections.

Imagine a situation in which two or more threads have access to a shared object which contains a counter variable declared like this:

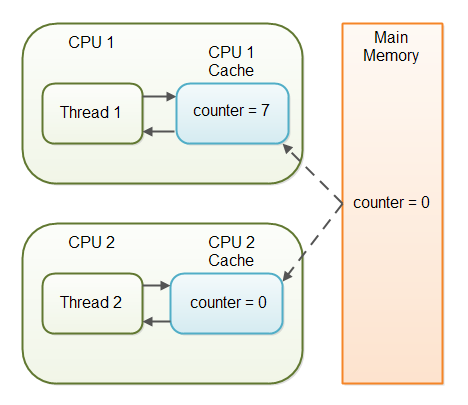
public class SharedObject {

public int counter = 0;

}

Imagine too, that only Thread 1 increments the counter variable, but both Thread 1 and Thread 2 may read the counter variable from time to time.

If the counter variable is not declared volatile there is no guarantee about when the value of the counter variable is written from the CPU cache back to main memory. This means, that the counter variable value in the CPU cache may not be the same as in main memory. This situation is illustrated here:



The problem with threads not seeing the latest value of a variable because it has not yet been written back to main memory by another thread, is called a "visibility" problem. The updates of one thread are not visible to other threads.

## The Java volatile Visibility Guarantee

The Java volatile keyword is intended to address variable visibility problems. By declaring the counter variable volatile all writes to the counter variable will be written back to main memory immediately. Also, all reads of the counter variable will be read directly from main memory.

Here is how the volatile declaration of the counter variable looks:

public class SharedObject {

public **volatile** int counter = 0;

}

Declaring a variable volatile thus *guarantees the visibility* for other threads of writes to that variable.

How Deadlock occurs?

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order. A Java multithreaded program may suffer from the deadlock condition because the **synchronized** keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object.

public class TestThread {

public static Object Lock1 = new Object();

public static Object Lock2 = new Object();

public static void main(String args[]) {

ThreadDemo1 T1 = new ThreadDemo1();

ThreadDemo2 T2 = new ThreadDemo2();

T1.start();

T2.start();

}

private static class ThreadDemo1 extends Thread {

public void run() {

synchronized (Lock1) {

System.out.println("Thread 1: Holding lock 1...");

try { Thread.sleep(10); }

catch (InterruptedException e) {}

System.out.println("Thread 1: Waiting for lock 2...");

synchronized (Lock2) {

System.out.println("Thread 1: Holding lock 1 & 2...");

}

}

}

}

private static class ThreadDemo2 extends Thread {

public void run() {

synchronized (Lock2) {

System.out.println("Thread 2: Holding lock 2...");

try { Thread.sleep(10); }

catch (InterruptedException e) {}

System.out.println("Thread 2: Waiting for lock 1...");

synchronized (Lock1) {

System.out.println("Thread 2: Holding lock 1 & 2...");

}

}

}

}

}

Output

Thread 1: Holding lock 1...

Thread 2: Holding lock 2...

Thread 1: Waiting for lock 2...

Thread 2: Waiting for lock 1...

The above program will hang forever because neither of the threads in position to proceed and waiting for each other to release the lock, so you can come out of the program by pressing CTRL+C.

## Deadlock Solution Example

Let's change the order of the lock and run of the same program to see if both the threads still wait for each other −

public class TestThread {

public static Object Lock1 = new Object();

public static Object Lock2 = new Object();

public static void main(String args[]) {

ThreadDemo1 T1 = new ThreadDemo1();

ThreadDemo2 T2 = new ThreadDemo2();

T1.start();

T2.start();

}

private static class ThreadDemo1 extends Thread {

public void run() {

synchronized (Lock1) {

System.out.println("Thread 1: Holding lock 1...");

try {

Thread.sleep(10);

} catch (InterruptedException e) {}

System.out.println("Thread 1: Waiting for lock 2...");

synchronized (Lock2) {

System.out.println("Thread 1: Holding lock 1 & 2...");

}

}

}

}

private static class ThreadDemo2 extends Thread {

public void run() {

synchronized (Lock1) {

System.out.println("Thread 2: Holding lock 1...");

try {

Thread.sleep(10);

} catch (InterruptedException e) {}

System.out.println("Thread 2: Waiting for lock 2...");

synchronized (Lock2) {

System.out.println("Thread 2: Holding lock 1 & 2...");

}

}

}

}

}

# **Java 8**

## Interface default and static methods.

Prior to java 8, [interface in java](https://beginnersbook.com/2013/05/java-interface/) can only have abstract methods.(declaration)

All the methods of interfaces are public & abstract by default. Java 8 allows the interfaces to have default and static methods. The reason we have default methods in interfaces is to allow the developers to add new methods to the interfaces without affecting the classes that implements these interfaces.

## Why default method?

For example, if several classes such as A, B, C and D implements an interface XYZInterface then if we add a new method to the XYZInterface, we have to change the code in all the classes(A, B, C and D) that implements this interface.

This is why in java 8, we have a new concept “default methods”. These methods can be added to any existing interface and we do not need to implement these methods in the implementation classes mandatorily, thus we can add these default methods to existing interfaces without breaking the code.

We can say that concept of default method is introduced in java 8 to add the new methods in the existing interfaces in such a way so that they are backward compatible. Backward compatibility is adding new features without breaking the old code

**Static methods** in interfaces are similar to the default methods except that we cannot override these methods in the classes that implements these interfaces.

interface Java8Interface{

/\* This is a default method so we need not to implement this method in the implementation classes \*/

default void newMethod(){

System.out.println("Newly added default method");

}

/\* This is a static method. Static method in interface is

\* similar to default method except that we cannot override

\* them in the implementation classes.

\*/

static void anotherNewMethod(){

System.out.println("Newly added static method");

}

/\* Already existing public and abstract method. We must need to implement this method in implementation classes.

\*/

void existingMethod(String str);

}

public class DefaultInterfaceMethodDemo implements Java8Interface {

// implementing abstract method

public void existingMethod(String str){

System.out.println("String is: "+str);

}

public static void main(String[] args) {

DefaultInterfaceMethodDemo obj = new DefaultInterfaceMethodDemo ();

//calling the default method of interface

obj.newMethod();

//calling the abstract method of interface

obj.existingMethod("Java 8 features");

//calling the static method of interface

Java8Interface.anotherNewMethod();

}

}

Output

Newly added default method

Newly added static method

String is: Java 8 features

**An Example of Diamond Problem with Default Methods**Java 8 is also supporting [**multiple inheritances**](http://javarevisited.blogspot.sg/2011/07/why-multiple-inheritances-are-not.html#axzz55Zuelntp) of classes?  
  
If you look closely, you will find that multiple inheritances of classes are **not supported** in Java 8, instead, the **compiler will do additional checks** to avoid ambiguity in calling default methods and Diamond problem, which could come if a class implements two interfaces that contain the default methods with the same name.

Class D extends A implements B, C {

}  
As shown below, this code will not compile in Java 8, because of ambiguity in calling the default method write() from a class, which extends both Poet and Writer interface.  
  
interface Poet

{

default void write()

{

System.out.println("Poet's default method");

}

}

**interface** Writer

{

**default** **void** write() {

System.***out***.println("Writer's default method");

}

}

**public** **class** Multitalented **implements** Poet, Writer

{

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

Multitalented john = **new** Multitalented();

john.write(); // which write method to call, Poet or Writer

}

}

Output: Compile Time Error : class Multitalented inherits unrelated defaults for write() from types Poet and Writer  
  
You can see that both Poet and Writer interface have a default method write() and when we have created a class Multitalented, which implements both Poet and Writer it gets the write() method from both super interface Poet and Writer.  
  
Now, the problem is not that we have implemented two interfaces with default methods of the same name. The Actual problems come when we created an object of a Multitalented class and called the write() method on its object

## How to avoid Diamond Problem With Default Methods in Java 8

In order to solve this error, you need to override the write() method in your implementation class i.e. class *Multitalented* here, this will remove the ambiguity, making the compiler happy enough to compile this class.  
  
**public** **class** Multitalented **implements** Poet, Writer

{

@Override

**public** **void** write(){

System.***out***.println("Writing stories now days");

}

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

Multitalented john = **new** Multitalented();

john.write();

}

}

# **Lambda Expression in Java 8**

Lambda expressions are similar to methods, it has arguments, a body and return type. They can also be called as anonymous methods. A method without name



One of the main features of lambda expressions is it enables passing a method as argument to another method. I see that because of this feature it is going to unravel umpteen programming possibilities.

## Functional Interfaces

An interface with no method is a marker interface. E.g Cloneable, Serializable

An interface with only one abstract method is a functional interface. ActionListener class is an example of functional interface.

We use anonymous class to implement ActionListener and in this kind of scenario, instead of using anonymous inner classes to implement, lambda expressions can be used. It will be simple and better compared to anonymous inner classes.

btn.addActionListener( new ActionListener() {

public void actionPerformed( ActionEvent event ) {

Toolkit.getDefaultToolkit().beep();

}

});

## Structure of a Lambda Expression

(Argument List) Arrow Token {Body }

## Example Lambda Expressions

() -> { System.out.printlns("Hello World!");}

(int a, int b) -> a + b

() -> { return 1; }

(String name) -> { System.out.println("Hello "+name); }

n -> n % 2 != 0

(**double** radius) -> **Math**.PI \* radius \* radius

radius -> { **return** **Math**.PI \* radius \* radius; }

radius -> { **System**.**out**.println(radius); **return** **Math**.PI \* radius \* radius; }

The first example's expression-based lambda body doesn't have to be placed between braces.

The second example converts the expression-based body to a statement-based body, in which return must be specified to return the expression's value.

The final example demonstrates multiple statements and cannot be expressed without the braces.

## Lambda Expression – Key Notes

* One important difference between anonymous inner class and lambda expression is, if we use ‘this’ it resolves to anonymous class but in the case of lambda expression, it resolves to the enclosing class.
* Lambda expression can have zero to any number of parameters.
* If there are no parameters to be passed, then an empty parentheses is given.
* Type of the passed parameter can be explicitly declared or can be taken from context.
* If the parameter’s type is inferred from the context then the parentheses need not be used.
* Lambda expression body can have zero to any number of statements.
* Body of expression should be enclosed in curly braces, if there is only one statement curly brace is not needed.

### 1. Lambda Expression Hello World

**public** **class** **LambdaHelloWorld** {

**interface** **HelloWorld** {

**String** hello(**String** name);

}

class Abc implements HelloWorld(){

String hello(String name){ syso(“Hello”+name);}

}

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

HelloWorld helloWorld = new Abc();

helloWorld.hello(“Joe”);

**HelloWorld** helloWorld = (**String** name) -> { **return** "Hello " + name; };

**System**.out.println(helloWorld.hello("Joe"));

}

}

**Sample code of Swing event handling.**  
import java.awt.BorderLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;

public class AnonymousListener {

public static void main(String[] args) {

JButton anonBtn = new JButton("Java Button");

//actionlistener using anonymous class

anonBtn.addActionListener(**new** **ActionListener**() {

**public** **void** actionPerformed(**ActionEvent** ae) {

**System**.out.println("Anonymous Click!");

}

});

//actionlistener using lambda expression

anonBtn.addActionListener(e -> { **System**.out.println("Lambda Click!");

});

JFrame frame = new JFrame("Lambda Expression Sample");

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

frame.add(anonBtn, BorderLayout.CENTER);

frame.pack();

frame.setVisible(true);

}

}

### 2. Lambda Expression with Runnable

Lambdas simplify the use of functional interfaces, which are annotated interfaces that each declare exactly one abstract method (although they can also declare any combination of default, static, and private methods). For example, the standard class library provides a java.lang.Runnable interface with a single abstract void run() method. This functional interface's declaration appears below:

@FunctionalInterface

**public** **interface** **Runnable**{

**public** **abstract** **void** run();

}

The class library annotates Runnable with @FunctionalInterface, which is an instance of the java.lang.FunctionalInterface annotation type. FunctionalInterface is used to annotate those interfaces that are to be used in lambda contexts.

A lambda doesn't have an explicit interface type. Instead, the compiler uses the surrounding context to infer which functional interface to instantiate when a lambda is specified--the lambda is *bound* to that interface. For example, suppose I specified the following code fragment, which passes the previous lambda as an argument to the java.lang.Thread class's Thread(Runnable target) constructor:

**new** **Thread**(() -> **System**.**out**.println("Hello"));

The compiler determines that the lambda is being passed to Thread(Runnable r) because this is the only constructor that satisfies the lambda: Runnable is a functional interface, the lambda's empty formal parameter list () matches run()'s empty parameter list, and the return types (void) also agree. The lambda is bound to Runnable.

Above lambda statement will be same as below code

public static void main(String[] args)

{

Runnable r = new Runnable()

{

@Override

public void run()

{

System.out.println("Hello");

}

};

new Thread(r).start();

}

Complete program of implementation of thread using anonymous class and lambda

**public** **class** RunnableLambdaExample {

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

RunnableLambdaExample obj=**new** RunnableLambdaExample();

// Main thread name

System.***out***.println("Main thread:"+Thread.*currentThread*().getName());

// Implementing Runnable using anonymous class implementation

Runnable r1=**new** Runnable() {

@Override

**public** **void** run() {

System.***out***.println("Thread with anonymous class :"+Thread.*currentThread*().getName());

}

};

Thread thread1 = **new** Thread(r1);

thread1.start(); //it will call the run method

// Pass Runnable in thread creation itself using anonymous class implementation.

Thread thread2 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

System.***out***.println("Thread with anonymous class :"+Thread.*currentThread*().getName());

}

});

thread2.start();

// Lambda expression with Runnable interface

Runnable r3 = () -> {

System.***out***.println("Thread with lambda:"+Thread.*currentThread*().getName());

System.***out***.println(Thread.*currentThread*().getPriority());

};

//new Thread(t2).start();

Thread thread3=**new** Thread(r3);

thread3.start();

}

}

### 3. Accessing Local and Class Variables in Lambda Expression

**public** **class** **LambdaVariableAccess** {

**public** **String** wildAnimal = "Lion";

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

**new** **LambdaVariableAccess**().lambdaExpression();

}

**public** **void** lambdaExpression(){

**String** domesticAnimal = "Dog";

**new** **Thread** (() -> {

**System**.out.println("Class Level: " + **this**.wildAnimal);

**System**.out.println("Method Level: " + domesticAnimal);

}).start();

}

}

### 4. Redeclaring Local Variables in Lambda Expression

**public** **class** JavaLocalVariableWithLambda

{

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

{

**int** limit = 10;

Runnable r = () -> {

**int** limit = 5;//variable can’t be re-declare in the scope.

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

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

};

}

}

Because limit is already present in the enclosing scope (the main() method), the lambda body's redefinition of limit (int limit = 5;) causes the compiler to report the error.

### 5. Modify Local Variables in the Lambda Expression

**public** **class** LambdaLocalVariableModification

{

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

{

**int** limit = 10;

Runnable r = () -> {

limit = 5; //variable defined in an enclosing scope main must be final or effectively final

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

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

};

}

}

A local variable or parameter that's defined outside a lambda body and referenced from the body must be marked final or considered effectively final (the variable cannot be assigned to after initialization). Attempting to modify an effectively final variable causes the compiler to report an error.

limit is effectively final. The lambda body's attempt to modify this variable causes the compiler to report an error. It does so because a final/effectively final variable will need to hang around until the lambda executes, which may not happen until long after the code in which the variable was defined returns. Non-final/non-effectively final variables no longer exist

### 6. Lambdas and the 'this' keyword

**public** **class** LambdaAndThis

{

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

{

LambdaAndThis obj = **new** LambdaAndThis();

System.out.printf("this = %s%n", **obj**);

obj.doWork();

}

**public** **void** doWork()

{

System.out.printf("this = %s%n", **this**);

Runnable r = **new** Runnable(){

@Override

**public** **void** run()

{

System.out.printf("this = %s%n", **this**);

}

};

**new** Thread(r).start();

**new** Thread(() -> System.out.printf("this = %s%n", **this**)).start();

}

}

this = LambdaAndThis@1db9742

this = LambdaAndThis@1db9742

this = LambdaAndThis$1@119bba2

this = LambdaAndThis@1db9742

The first line & second line shows LambdaAndThis’s this reference, the third line shows a different this reference in the new Runnable scope, and the fourth output line shows the this reference in a lambda context. The fourth and first lines match because the lambda's scope is nested inside the doWork() method; this has the same meaning throughout this method.

### 7. Lambdas and exceptions

A lambda body is not allowed to throw more exceptions than those are specified in the throws clause of the functional interface method. If a lambda body throws an exception, the functional interface method's throws clause must declare the same exception type or its supertype.

**import** java.awt.AWTException;

**import** java.io.IOException;

@FunctionalInterface

**interface** Work

{

**void** doSomething() **throws** IOException;

}

**public** **class** LambdaAndException

{

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

{

Work work = () -> { **throw** **new** IOException(); };

work.doSomething();

work = () -> { **throw** **new** AWTException(""); };

}

}

A Work functional interface whose doSomething() method is declared to throw java.io.IOException. The main() method assigns a lambda that throws IOException to work, which is okay because IOException is listed in doSomething()'s throws clause.

### 8) Predefined functional interfaces

You might find yourself repeatedly creating similar functional interfaces. For example, you might create a CheckConnection functional interface with a boolean isConnected(Connection c) method and a CheckAccount functional interface with a boolean isPositiveBalance(Account acct) method. This is wasteful.

The previous examples expose the abstract concept of a *predicate* (a Boolean-valued function). Anticipating such patterns, Oracle provides the java.util.function package of commonly-used functional interfaces. For example, this package's Predicate<T> functional interface can be used in place of CheckConnection and CheckAccount.

Predicate<T> provides a boolean test(T t) method that evaluates this predicate on its argument (t), returning true when t matches the predicate, and returning false otherwise. Notice that test() provides the same kind of parameter list as isConnected() and isPositiveBalance(). Also, notice that they all have the same return type (boolean).

import java.util.ArrayList;

import java.util.List;

import java.util.function.Predicate;

class Account

{

private int id, balance;

Account(int id, int balance)

{

this.balance = balance;

this.id = id;

}

int getBalance()

{

return balance;

}

int getID()

{

return id;

}

void print()

{

System.out.printf("Account: [%d], Balance: [%d]%n", id, balance);

}

}

public class LambdaPredicates

{

static List<Account> accounts;

public static void main(String[] args)

{

accounts = new ArrayList<>();

accounts.add(new Account(1000, 200));

accounts.add(new Account(2000, -500));

accounts.add(new Account(3000, 0));

accounts.add(new Account(4000, -80));

accounts.add(new Account(5000, 1000));

// Print all accounts

System.***out***.println("Print all accounts");

*printAccounts*(account -> **true**);

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

// Print all accounts with negative balances.

System.***out***.println("Print all accounts with negative balance");

*printAccounts*(account -> account.getBalance() < 0);

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

// Print all accounts whose id is greater than 2000 and less than 5000.

System.***out***.println("Print all accounts whose id is greater than 2000 and less than 5000");

*printAccounts*(account -> account.getID() > 2000 && account.getID() < 5000);

}

static void printAccounts(Predicate<Account> tester)

{

for (Account account: accounts)

if (tester.test(account))

account.print();

}

}

**Output:**

Print all accounts

Account: [1000], Balance: [200]

Account: [2000], Balance: [-500]

Account: [3000], Balance: [0]

Account: [4000], Balance: [-80]

Account: [5000], Balance: [1000]

Print all accounts with negative balance

Account: [2000], Balance: [-500]

Account: [4000], Balance: [-80]

Print all accounts whose id is greater than 2000 and less than 5000

Account: [3000], Balance: [0]

Account: [4000], Balance: [-80]

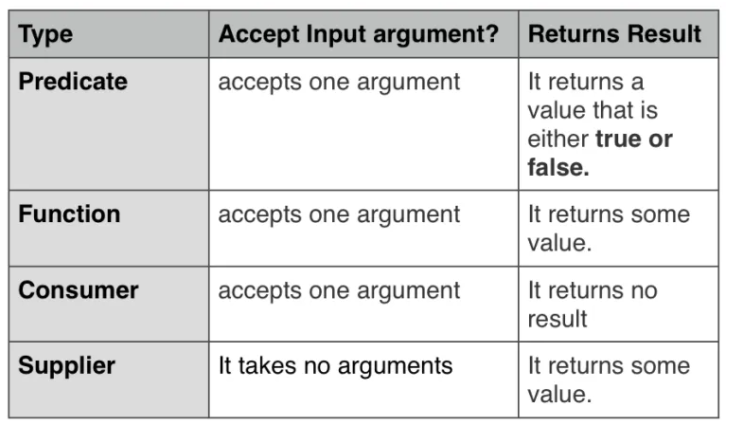
## Functional Interfaces: Predicate, Consumer, Function, and Supplier

Functional interfaces provide target types for lambda expressions and method references. Each functional interface has a single abstract method, called functional method for that functional interface, to which the lambda expression’s parameter and return types are matched or adapted.

There are around **40+ functional interfaces** under **[java.util.function](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html" \t "_blank)** package.In this blog, we will discuss the important ones: **Predicate, Consumer, Function, and Supplier.**

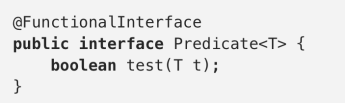
**Complete list of Function interface**

<https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>



#### Predicate

A predicate is a statement that may be true or false depending on the values of its variables. It can be thought of as a function that returns a value that is either true or false.



Predicate<String> predFunc = t -> t.length() > 5;

System.***out***.println("Is Programming String has length > 5 : "+predFunc.test("Programming"));

System.***out***.println("Is length of the ‘Dance’ > 5 :"+predFunc.test("Dance"));

// Predicate use to filter the collection

List<String> list1= **new** ArrayList<>();

list1.add("Pizza");

list1.add("Burger");

list1.add("oats");

list1.add("Almonds");

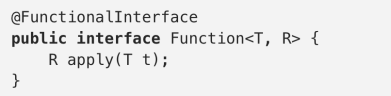
list1.add("Muli");

List<String> filteredString= list1.stream().filter(t->t.length()>=5).collect(Collectors.*toList*());

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

#### **Function**

This functional interface represents a function that accepts one argument and produces a result. One use, for example, it’s to convert or transform from one object to another. Since it’s a functional interface, you can pass a lambda expression wherever a Function is expected.  
The input parameter type and the return type of the method can either be same or different.



Function<Integer,Integer> funcInterface = t -> t\*2;

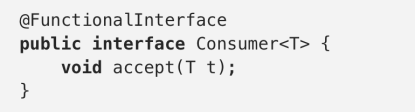
System.***out***.println(funcInterface.apply(5));

Function<Integer,String> funcInterface1 = t -> t.toString();

System.***out***.println(funcInterface1.apply(5));

#### **Consumer**

This functional interface represents an operation that accepts a single input argument and returns no result. The real outcome is the side-effects it produces. Since it’s a functional interface, you can pass a lambda expression wherever a Consumer is expected.



**class** Emp{

**private** String name;

**public** Emp(String s) {

**this**.name=s;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

Emp emp=**new** Emp("Radha");

Consumer<Emp> updateName= s->s.setName("Radhe");

updateName.accept(emp);

System.***out***.println("Consumer update name :"+emp.getName());

#### **Supplier**

This functional interface does the opposite of the Consumer, it takes no arguments but it returns some value. It may return different values when it is being called more than once. Since it’s a functional interface, you can pass a lambda expression wherever a Supplier is expected.



Supplier<Integer> supplierFunc = () -> 20;

System.***out***.println(supplierFunc.get());

Complete program:

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.function.Consumer;

**import** java.util.function.Function;

**import** java.util.function.Predicate;

**import** java.util.function.Supplier;

**import** java.util.stream.Collectors;

**public** **class** FunctionalInterfacesDemo {

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

Predicate<String> predFunc = t -> t.length() > 5;

System.***out***.println("Is Programming > 5 : "+predFunc.test("Programming"));

System.***out***.println("Is Dance > 5 :"+predFunc.test("Dance"));

// Predicate use to filter the collection

List<String> list1= **new** ArrayList<>();

list1.add("Pizza");

list1.add("Burger");

list1.add("oats");

list1.add("Almonds");

list1.add("Muli");

List<String> filteredString= list1.stream().filter(t->t.length()>=5).collect(Collectors.*toList*());

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

// Function :This functional interface represents a function that accepts one argument

// and produces a result. One use, for example, it’s to convert or transform from one object to another.

Function<Integer,Integer> funcInterface = t -> t\*2;

System.***out***.println("Function interface with return t\*2 in integer :"+funcInterface.apply(5));

Function<Integer,String> funcInterface1 = t -> t.toString();

System.***out***.println("Function interface with return String :"+funcInterface1.apply(5));

/\* Consumer : This functional interface represents an operation that accepts a

\* single input argument and returns no result.

\* Since it’s a functional interface, you can pass a lambda expression wherever a Consumer

\* is expected.

\*/

**class** Emp{

**private** String name;

**public** Emp(String s) {

**this**.name=s;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

Emp emp=**new** Emp("Radha");

Consumer<Emp> updateName= s->s.setName("Radhe");

updateName.accept(emp);

System.***out***.println("Consumer update name :"+emp.getName());

/\*Supplier : This functional interface does the opposite of the Consumer,

\* it takes no arguments but it returns some value.

\* It may return different values when it is being called more than once.

\* Since it’s a functional interface, you can pass a lambda expression wherever a Supplier is expected.

\*/

Supplier<Integer> supplierFunc = () -> 20;

System.***out***.println("Supplier value :"+supplierFunc.get());

}

}

Output :

Is Programming > 5: true

Is Dance > 5: false

[Pizza, Burger, Almonds]

Function interface with return t\*2 in integer :10

Function interface with return String :5

Consumer update name :Radhe

Supplier value :20

# **Java 8 method references, double colon (::) operator**

In Java 8, the double colon (::) operator is called method references.

Anonymous class to print a list.

List<String> list = Arrays.asList("node", "java", "python", "ruby");

list.forEach(new Consumer<String>() { *// anonymous class*

@Override

public void accept(String str) {

System.out.println(str);

}

});

Anonymous class -> Lambda expressions.

List<String> list = Arrays.asList("node", "java", "python", "ruby");

list.forEach(str -> System.out.println(str)); *// lambda*

Lambda expressions -> Method references.

List<String> list = Arrays.asList("node", "java", "python", "ruby");

list.forEach(System.out::println); *// method references , out is PrintStream class*

There are four kinds of method references:

* Reference to a static method ClassName::staticMethodName
* Reference to an instance method of a particular object Object::instanceMethodName
* Reference to an instance method of an arbitrary object of a particular type ContainingType::methodName–
* Reference to a constructor ClassName::new

## 1. Static method

Lambda expression.

(args) -> ClassName.staticMethodName(args)

Method Reference.

ClassName::staticMethodName

1.1 This example prints a list of Strings, method reference to a static method SimplePrinter::print.

import java.util.Arrays;

import java.util.List;

import java.util.function.Consumer;

public class MethodReferenceStatic {

public static void main(String[] args) {

List<String> list = Arrays.asList("A", "B", "C");

// anonymous class

list.forEach(new Consumer<String>() {

@Override

public void accept(String x) {

SimplePrinter.print(x);

}

});

// lambda expression

list.forEach(x -> SimplePrinter.print(x));

// method reference

list.forEach(SimplePrinter::print);

}

}

class SimplePrinter {

public static void print(String str) {

System.out.println(str);

}

}

1.2 This example converts a list of Strings into a list of Integers, method reference to a static method Integer::parseInt.

Integer.java

public static int parseInt(String s) throws NumberFormatException {

return parseInt(s,10);

}

**MethodReferenceStatic1.java**

import java.util.Arrays;

import java.util.List;

import java.util.function.Function;

import java.util.stream.Collectors;

public class MethodReferenceStatic1 {

public static void main(String[] args) {

List<String> list = Arrays.asList("1", "2", "3");

// anonymous class

List<Integer> collect1 = list.stream()

.map(new Function<String, Integer>() {

@Override

public Integer apply(String s) {

return Integer.parseInt(s);

}

})

.collect(Collectors.toList());

// lambda expression

List<Integer> collect2 = list.stream()

.map(s -> Integer.parseInt(s))

.collect(Collectors.toList());

// method reference

List<Integer> collect3 = list.stream()

.map(Integer::parseInt)

.collect(Collectors.toList());

System.***out***.println("Output using anonymous class :"+collect1);

System.***out***.println("Output using Lambda :"+collect2);

System.***out***.println("Output using method reference :"+collect3);

}

}

## 2. Reference to an instance method of a particular object

Lambda expression.

(args) -> object.instanceMethodName(args)

Method Reference.

object::instanceMethodName

2.1 This example sorts a list of Employee by salary. We can reference to an instance method compareBySalary of a particular object ComparatorProvider.

import java.math.BigDecimal;

import java.util.Arrays;

import java.util.List;

public class Java8MethodReference2 {

public static void main(String[] args) {

List<Employee> list = Arrays.asList(

new Employee("Kunal", 38, BigDecimal.valueOf(38000)),

new Employee("Shyam", 5, BigDecimal.valueOf(1000)),

new Employee("Riya", 25, BigDecimal.valueOf(25000)),

new Employee("Saloni", 99, BigDecimal.valueOf(99999)));

ComparatorProvider provider = new ComparatorProvider();

// anonymous class

/\*list.sort(new Comparator<Employee>() {

@Override

public int compare(Employee o1, Employee o2) {

return provider.compareBySalary(o1, o2);

}

});\*/

// lambda

// list.sort((o1, o2) -> provider.compareBySalary(o1, o2));

// method reference

list.sort(provider::compareBySalary);

list.forEach(x -> System.out.println(x));

}

}

class ComparatorProvider {

public int compareByAge(Employee o1, Employee o2) {

return o1.getAge().compareTo(o2.getAge());

}

public int compareByName(Employee o1, Employee o2) {

return o1.getName().compareTo(o2.getName());

}

public int compareBySalary(Employee o1, Employee o2) {

return o1.getSalary().compareTo(o2. getSalary ());

}

}

import java.math.BigDecimal;

public class Employee {

String name;

Integer age;

BigDecimal salary;

// getters, setters, constructor, toString

}

## 3. Reference to an instance method of an arbitrary object of a particular type.

Lambda expression.

*// arg0 is the first argument*

(arg0, rest\_of\_args) -> arg0.methodName(rest\_of\_args)

*// example, assume a and b are String*

(a, b) -> a.compareToIgnoreCase(b)

Method Reference.

*// first argument type*

arg0\_Type::methodName

*// arg0 is type of ClassName*

ClassName::methodName

*// example, a is type of String*

String::compareToIgnoreCase

To understand this better lets first learn BiFunction Interface

In Java 8, *[BiFunction](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)* is a functional interface; it takes two arguments and returns an object.

BiFunction.java

@FunctionalInterface

public interface BiFunction<T, U, R> {

R apply(T t, U u);

}

* T – Type of the first argument to the function.
* U – Type of the second argument to the function.
* R – Type of the result of the function.

## BiFunction<T, U, R>

This example takes two Integers and returns an Integer, Double or List

import java.util.Arrays;

import java.util.List;

import java.util.function.BiFunction;

public class Java8BiFunction1 {

public static void main(String[] args) {

// takes two Integers and return an Integer

BiFunction<Integer, Integer, Integer> func = (x1, x2) -> x1 + x2;

Integer result = func.apply(2, 3);

System.out.println(result); // 5

// take two Integers and return an Double

BiFunction<Integer, Integer, Double> func2 = (x1, x2) -> Math.pow(x1, x2);

Double result2 = func2.apply(2, 4);

System.out.println(result2); // 16.0

// take two Integers and return a List<Integer>

BiFunction<Integer, Integer, List<Integer>> func3 = (x1, x2) -> Arrays.asList(x1 + x2);

List<Integer> result3 = func3.apply(2, 3);

System.out.println(result3);

}

}

## Example : Reference to an instance method of an arbitrary object of a particular type.

import java.util.function.BiPredicate;

import java.util.function.Function;

public class Java8MethodReference3a {

public static void main(String[] args) {

// lambda

int result = playOneArgument("Python", x -> x.length()); // 6

// method reference

int result2 = playOneArgument("Python", String::length); // 6

// lambda

Boolean result3 = playTwoArgument("Python ", "y", (a, b) -> a.contains(b)); // true

// method reference

Boolean result4 = playTwoArgument("Python", "y", String::contains); // true

// lambda

Boolean result5 = playTwoArgument("Python", "y", (a, b) -> a.startsWith(b)); // false

// method reference

Boolean result6 = playTwoArgument("Python", "y", String::startsWith); // false

System.out.println(result6);

}

static <R> R playOneArgument(String s1, Function<String, R> func) {

return func.apply(s1);

}

static Boolean playTwoArgument(String s1, String s2, BiPredicate<String, String> func) {

return func.test(s1, s2);

}

}

## 4. Reference to a constructor.

Lambda expression.

(args) -> new ClassName(args)

Method Reference.

ClassName::new

4.1 Reference to a default constructor.

**import** java.math.BigDecimal;

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.function.Supplier;

**public** **class** MethodReferenceConstrcutorDemo {

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

// lambda

Supplier<Map> obj1 = () -> **new** HashMap(); // default HashMap() constructor

Map map1 = obj1.get();

// method reference

Supplier<Map> obj2 = HashMap::**new**;

Map map2 = obj2.get();

// lambda

Supplier<Invoice> obj3 = () -> **new** Invoice(); // default Invoice() constructor

Invoice invoice1 = obj3.get();

// method reference

Supplier<Invoice> obj4 = Invoice::**new**;

Invoice invoice2 = obj4.get();

}

}

**class** Invoice {

String no;

BigDecimal unitPrice;

Integer qty;

**public** Invoice() {

}

**public** String getNo() {

**return** no;

}

**public** **void** setNo(String no) {

**this**.no = no;

}

**public** BigDecimal getUnitPrice() {

**return** unitPrice;

}

**public** **void** setUnitPrice(BigDecimal unitPrice) {

**this**.unitPrice = unitPrice;

}

**public** Integer getQty() {

**return** qty;

}

**public** **void** setQty(Integer qty) {

**this**.qty = qty;

}

}

4.2 Reference to a constructor which accepts an argument – Invoice(BigDecimal unitPrice)

**import** java.math.BigDecimal;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.function.Function;

**public** **class** MethodReferenceConstructorDemo2 {

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

List<BigDecimal> list = Arrays.*asList*(

BigDecimal.*valueOf*(9.99),

BigDecimal.*valueOf*(2.99),

BigDecimal.*valueOf*(8.99));

// lambda

// List<Invoice> invoices = fakeInvoice(list, (unit) -> new Invoice1(unit));

// method reference

List<Invoice1> invoices = *fakeInvoice*(list, Invoice1::**new**);

invoices.forEach(System.***out***::println);

}

**static** List<Invoice1> fakeInvoice(List<BigDecimal> list, Function<BigDecimal, Invoice1> func) {

List<Invoice1> result = **new** ArrayList<>();

**for** (BigDecimal unit : list) {

result.add(func.apply(unit));

}

**return** result;

}

}

**class** Invoice1 {

String no;

BigDecimal unitPrice;

Integer qty;

**public** Invoice1() {

}

**public** Invoice1(BigDecimal bd) {

**this**.unitPrice = bd;

}

**public** String getNo() {

**return** no;

}

**public** **void** setNo(String no) {

**this**.no = no;

}

**public** BigDecimal getUnitPrice() {

**return** unitPrice;

}

**public** **void** setUnitPrice(BigDecimal unitPrice) {

**this**.unitPrice = unitPrice;

}

**public** Integer getQty() {

**return** qty;

}

**public** **void** setQty(Integer qty) {

**this**.qty = qty;

}

**public** String toString() {

**return** **this**.getUnitPrice().toString();

}

}

Output

9.99

2.99

8.99