**Java interview questions and answer**

### what is consumer

ans:-It is a functional interface defined in java.util.function package. It contains an abstract accept() and a default andThen() method. It can be used as the assignment target for a lambda expression or method reference.

The Consumer Interface accepts a single argument and does not return any result.

1. **import** java.util.function.Consumer;
2. **public** **class** ConsumerInterfaceExample {
3. **static** **void** printMessage(String name){
4. System.out.println("Hello "+name);
5. }
6. **static** **void** printValue(**int** val){
7. System.out.println(val);
8. }
9. **public** **static** **void** main(String[] args) {
10. // Referring method to String type Consumer interface
11. Consumer<String> consumer1 = ConsumerInterfaceExample::printMessage;
12. consumer1.accept("John");   // Calling Consumer method
13. // Referring method to Integer type Consumer interface
14. Consumer<Integer> consumer2 = ConsumerInterfaceExample::printValue;
15. consumer2.accept(12);   // Calling Consumer method
16. }
17. }

Output:

Hello John

12

2.what is supplier?

## Ans:- **Supplier Interface :-**

Supplier ( java.util.function.Supplier ) is a functional interface that has one abstract method declared in it. It represents an operation by which you can generate new values in the stream.

***T  get (  ) :-*** The abstract method ***get (  )***is a functional method. It does not accept any argument but return newly generated values ***( T )*** in the stream.

There are four specialised ( primitive specific ) form of Java 8 Supplier interface, which can be used in case of generating **int**, **long**, **double and boolean**. They are as follows :-

1. [**IntSupplier**](https://codedestine.com/java-8-intsupplier-interface/)**interface (**[**JavaDocs**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntSupplier.html)**) :-** It represents an operation by which you can generate new int values in the stream.
2. [**LongSupplier**](https://codedestine.com/java-8-longsupplier-interface/)**interface (**[**JavaDocs**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongSupplier.html)**) :-** It represents an operation by which you can generate new long values in the stream.
3. [**DoubleSupplier**](https://codedestine.com/java-8-doublesupplier-interface/)**interface (**[**JavaDocs**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleSupplier.html)**) :-** It represents an operation by which you can generate new double values in the stream.
4. [**BooleanSupplier**](https://codedestine.com/java-8-booleansupplier-interface/)**interface (**[**JavaDocs**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BooleanSupplier.html)**) :-** It represents an operation by which you can generate new boolean values in the stream.

**mployee ( POJO ) :-**Used in our example.

[?](https://codedestine.com/java-8-supplier-interface/)

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | public class Employee {       private long empId;     private String name;     private int age;     private String designation;       public Employee(long empId, String name, int age, String designation) {       super();       this.empId = empId;       this.name = name;       this.age = age;       this.designation = designation;     }       public long getEmpId() {       return empId;     }     public void setEmpId(long empId) {       this.empId = empId;     }     public String getName() {       return name;     }     public void setName(String name) {       this.name = name;     }     public int getAge() {       return age;     }     public void setAge(int age) {       this.age = age;     }     public String getDesignation() {       return designation;     }     public void setDesignation(String designation) {       this.designation = designation;     }    } |

## **Example – 1 :-**

This example will show you, how to create and call different methods of a supplier interface.

[?](https://codedestine.com/java-8-supplier-interface/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public static void main(String[] args){      Supplier<Employee> supplier = ( ) -> { return new Employee(1,"Robert",35,"Manager"); };      Employee emp = supplier.get();    System.out.println(emp.getDesignation());  } |

## **Result :-**

Manager

### 3.what is function in java 8?

### Function interface is used to do the transformation.It can accepts one argument and produces a result.

public class Main {

   public static void main(String args[]) {

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

      numList.add(78);

      numList.add(10);

      Function<Integer, Integer> fun = i -> i / 2;

      numList.stream().map(fun).forEach(System.out::println);

   }

}

4.what is predicate in java?

On the other side, Predicate can also accept only one argument but it can only return boolean value.

public class Main {

   public static void main(String args[]) {

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

      numList.add(5);

      numList.add(10);

      Predicate<Integer> pred = i -> i > 5;

      numList.stream().filter(pred).forEach(i -> System.out.println(i));

   }

}

5.what is caching in jpa?

How to configure spring cache in **Service methods** in conjunction with **@Cacheable**, **@CacheEvict** annotations, so that when a record added/updated (write) to database it will be reflected in the caches of findAll, findById (read) methods

i’m using a delivery record example to illustrate the cashing behaviour.

@Slf4j  
@Service  
**public class** DeliveryService {  
 @Autowired  
 DeliveryRepository **deliveryRepository**;  
  
 @Cacheable(**"deliveries"**)  
 **public** List<DeliveryDto> findAll(){  
 ***log***.info(**"DeliveryService: findAll"**);  
 List<Delivery> deliveries = **deliveryRepository**.findAll();  
 List<DeliveryDto> deliveryDtos = **new** ArrayList<>();  
 **for** (Delivery delivery:deliveries){  
 DeliveryDto deliveryDto = **new** DeliveryDto();  
 *copyProperties*(delivery, deliveryDto);  
 deliveryDtos.add(deliveryDto);  
 }  
 **return** deliveryDtos;  
 }  
  
 @Cacheable(**"delivery"**)  
 **public** DeliveryDto findById(Long id){  
 ***log***.info(**"DeliveryService: findById"**);  
 Delivery delivery = **deliveryRepository**.findById(id).orElse(**null**);  
 DeliveryDto deliveryDto = **new** DeliveryDto();  
 *copyProperties*(delivery,deliveryDto);  
 **return** deliveryDto;  
 }  
  
 @Caching(evict = {  
 @CacheEvict(value=**"delivery"**, allEntries=**true**),  
 @CacheEvict(value=**"deliveries"**, allEntries=**true**)})  
 **public** Delivery saveOrUpdate(DeliveryDto deliveryDto){  
 ***log***.info(**"DeliveryService: saveOrUpdate, {}"**, deliveryDto.getPickupName());  
 Delivery delivery = Delivery.*builder*()  
 .pickupName(deliveryDto.getPickupName())  
 *...* .build();  
 **if**(deliveryDto.getId()!=**null**){  
 delivery.setId(deliveryDto.getId());  
 }  
 **return deliveryRepository**.save(delivery);  
 }  
}

Repository class

**package** com.buddhi.repository;  
  
**import** com.buddhi.model.Delivery;  
**import** org.springframework.data.jpa.repository.JpaRepository;  
**import** org.springframework.stereotype.Repository;  
  
@Repository  
**public interface** DeliveryRepository **extends** JpaRepository<Delivery, Long> {  
}

3.why override equals and hashcode ?

## 1. Uses of hashCode() and equals() Methods

1. equals(Object otherObject) – verifies the equality of two objects. Its default implementation simply checks the object references of two objects to verify their equality.  
   By default, two objects are equal if and only if they are refer to the same memory location. Most Java classes override this method to provide their own comparison logic.
2. hashcode() – returns a unique integer value for the object in runtime.  
   By default, integer value is derived from memory address of the object in heap (but it’s not mandatory).  
   The object’s hash code is used for determining the index location, when this object needs to be stored in some [HashTable](https://en.wikipedia.org/wiki/Hash_table) like data structure.

#### 2.1. The default behavior of Employee class

Let’s take an example where your application has Employee object. Let us create a minimal possible structure of Employee class:

**public** **class** Employee

{

**private** Integer id;

**private** String firstname;

**private** String lastName;

**private** String department;

*//Setters and Getters*

}

Above Employee class has some fundamental attributes and their accessor methods. Now consider a simple situation where you need to [**compare two Employee objects**](https://howtodoinjava.com/java/collections/java-comparable-interface/). Both employee objects have the same id.

**public** **class** EqualsTest {

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

Employee e1 = **new** Employee();

Employee e2 = **new** Employee();

e1.setId(100);

e2.setId(100);

System.out.println(e1.equals(e2)); *//false*

}

}

No prize for guessing. The above method will print “false.”

But is it correct after knowing that both objects represent the same employee? In a real-time application, this should return *true*.

#### 2.2. Should we override only equals() method?

To achieve correct application behavior, we need to override equals() method as below:

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

**if**(o == **null**)

{

**return** false;

}

**if** (o == **this**)

{

**return** true;

}

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

{

**return** false;

}

Employee e = (Employee) o;

**return** (**this**.getId() == e.getId());

}

Add this method to the Employee class, and EqualsTest will start returning "true".

So are we done? Not yet. Let’s test the above-modified Employee class again in a different way.

**import** java.util.HashSet;

**import** java.util.Set;

**public** **class** EqualsTest

{

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

{

Employee e1 = **new** Employee();

Employee e2 = **new** Employee();

e1.setId(100);

e2.setId(100);

*//Prints 'true'*

System.out.println(e1.equals(e2));

Set<Employee> employees = **new** HashSet<Employee>();

employees.add(e1);

employees.add(e2);

System.out.println(employees); *//Prints two objects*

}

}

The above example prints two objects in the second print statement.

If both employee objects have been equal, in a Set which stores unique objects, there must be only one instance inside HashSet because both objects refer to the same employee. What is it we are missing??

#### 2.3. Overriding hashCode() is necessary

We are missing the second important method hashCode(). As java docs say, if we override equals() then we **must** override hashCode(). So let’s add another method in our Employee class.

@Override

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

{

**final** **int** PRIME = 31;

**int** result = 1;

result = PRIME \* result + getId();

**return** result;

}

Once the above method is added in Employee class, the second statement starts printing only a single object in the second statement and **thus validating the true equality of e1 and e2**.

* 5. 1 . Why is Java so popular?
* One of the biggest reasons why Java is so popular is the **platform independence**. Programs can run on several different types of computer; as long as the computer has a Java Runtime Environment (JRE) installed, a Java program can run on it.

1. What is platform independence?

**Software that can run on a variety of hardware platforms or software architectures**. Platform-independent software can be used in many different environments, requiring less planning and translation across an enterprise.

 What is bytecode?

Bytecode in Java is an intermediate machine-independent code. It is a set of instructions for Java Virtual Machine and it acts pretty similar to the assembler in C++. In general, bytecode is a code that lies between low-level and high-level language. The bytecode is not processed by the processor. It is processed by the Java Virtual Machine (JVM). The job of the JVM is to call all the required resources to compile the Java program and make the bytecode independent. It is the biggest reason why java is known as a platform-independent language. The intermediate code can run on any of the platforms such as Windows, macOS, and Linux.

============

 What is the role for a classloader in Java?

Class loaders are responsible for **loading Java classes dynamically to the JVM (Java Virtual Machine) during runtime**. They're also part of the JRE (Java Runtime Environment). Therefore, the JVM doesn't need to know about the underlying files or file systems in order to run Java programs thanks to class loaders.

What are Wrapper classes?

The classes that assist us in creating and utilizing the object of the primitive data types are known as the java wrapper classes. For example, Character is a wrapper class for char, Float for float, Integer for int, and so on.

 Why do we need Wrapper classes in Java?

Wrapper Class will **convert primitive data types into objects**. The objects are necessary if we wish to modify the arguments passed into the method (because primitive types are passed by value). The classes in java. util package handles only objects and hence wrapper classes help in this case also.3

What are the different ways of creating Wrapper class instances?

**Methods:**

* Using the constructor of the wrapper class.
* Using the valueOf() method provided by the Wrapper classes.
* Using concept of AutoBoxing.

## Creating Wrapper Objects

* **Using a Wrapper Class Constructor**  
  We can create a wrapper object using the wrapper class and its constructor by passing the value to it.

Syntax:

ClassName object = new ClassName(argument);

Example:

Integer number = new Integer(77);*//int*

Integer number2 = new Integer("77");*//String*

Float number3 = new Float(77.0);*//double argument*

Float number4 = new Float(77.0f);*//float argument*

Float number5 = new Float("77.0f");*//String*

Character c1 = new Character('S');*//Only char constructor*

Character c2 = new Character(1234);*//COMPILER ERROR*

Boolean b = new Boolean(true);*//value stored - true*

However, the way of creating an instance of wrapper classes using constructor is deprecated as of the latest version of JDK. This is because each time new memory is allocated in the heap when we create an object with the help of the constructor. Also, the constructor Character(char) has been deprecated since JDK version 9.

* **Using Wrapper class only (instead of the primitive type)**  
  We can create without using constructor as well by using the wrapper class instead of the primitive type to create a wrapper object without passing the value to the constructor like we did earlier method. And to get the value, we can print the particular object.

Syntax:

ClassName object = value;

*// of primitive data type associated with the wrapper class.*

Example:

public class CreatingWrapperObject {

public static void main(String[] args) {

*//Creating the object using the wrapper class*

without passing the value to the constructor

Integer intValue = 10; *// object intValue of type Integer will store the value 10 as int*

Double doubleValue = 8.89;

Character charValue = 'S';

*//Printing the values using the created objects*

System.out.println(intValue);

System.out.println(doubleValue);

System.out.println(charValue);

}

}

Output:

10

8.89

S

Another example where we are converting an Integer to a String, and using the length() method to calculate the length of the "string":

public class CreatingWrapperObject2 {

public static void main(String[] args) {

*//Creating the object using the Wrapper class*

Integer intValue = 1000;

*//Converting the integer value to String and assigning it to stringObject*

String stringObject = intValue.toString(); *//toString() method used for the conversion*

*//Printing the length of the String using length() method*

System.out.println(stringObject.length());

}

Output:

4

* **Using valueOf Static methods**  
  By using valueOf Static method, a Wrapper object can be created.

Syntax:

ClassName object = ClassName.valueOf(argument);

Example:

Integer hundred = Integer.valueOf("100");

*//100 is stored in variable. Here, Integer.valueOf(String str) is used.*

Integer seven = Integer.valueOf("111", 2);

*//binary 111 is converted to 7. Here, Integer.valueOf(String str, int base) is used.*

**Note:**

The difference in using the other methods and valueOf() static method is - By using the Constructor or Wrapper Class method we will always create a new object which will allocate a new memory in the heap each time, while using valueOf() static method, it may return a cached value with-in a range.

=====

What are differences in the two ways of creating Wrapper classes?

What are differences in the two ways of creating Wrapper Classes? The difference is that **using the Constructor you will always create a new object, while using valueOf() static method, it may return you a cached value with-in a range**.

What is auto boxing?

## Autoboxing

Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an *int*to an *Integer*, a *double*to a *Double*, and so on.

### Autoboxing Example

This program demonstrates the *auto-boxing* of all the *primitive types*.

/\*\*

\* Class to demonstrate auto-boxing the primitive types.

\* @author javaguides.net

\*

\*/

public class AutoBoxingExample {

public static void main(String[] args) {

byte b = 100; //Primitive byte data

Byte B = b; //Auto-Boxing of byte data

System.out.println(B);

short s = 100; //Primitive short data

Short S = s; //Auto-Boxing of short data

System.out.println(S);

int i = 200; //Primitive int Data

Integer I = i; //Auto-Boxing of int data

System.out.println(I);

long l = 250;

Long L = l;

System.out.println(L);

float f = 120L;

Float F = f;

System.out.println(F);

double d = 18.58;

Double D = d;

System.out.println(D);

boolean bln = false;

Boolean BLN = bln;

System.out.println(BLN);

char c = 'C';

Character C = c;

System.out.println(C);

}

}

Output:

100

100

200

250

120.0

18.58

false

C

 What are the advantages of auto boxing?

Autoboxing and unboxing **lets developers write cleaner code, making it easier to read**. The technique lets us use primitive types and Wrapper class objects interchangeably and we do not need to perform any typecasting explicitly.

What is casting?

Type casting is **when you assign a value of one primitive data type to another type**. In Java, there are two types of casting: Widening Casting (automatically) - converting a smaller type to a larger type size. byte -> short -> char -> int -> long -> float -> double.

## **Widening Casting**

Widening casting is done automatically when passing a smaller size type to a larger size type:

### Example

public class Main {

public static void main(String[] args) {

int myInt = 9;

double myDouble = myInt; // Automatic casting: int to double

System.out.println(myInt); // Outputs 9

System.out.println(myDouble); // Outputs 9.0

}

}

[Try it Yourself »](https://www.w3schools.com/java/tryjava.asp?filename=demo_casting_wide)

## **Narrowing Casting**

Narrowing casting must be done manually by placing the type in parentheses in front of the value:

### Example

public class Main {

public static void main(String[] args) {

double myDouble = 9.78d;

int myInt = (int) myDouble; // Manual casting: double to int

System.out.println(myDouble); // Outputs 9.78

System.out.println(myInt); // Outputs 9

}

}

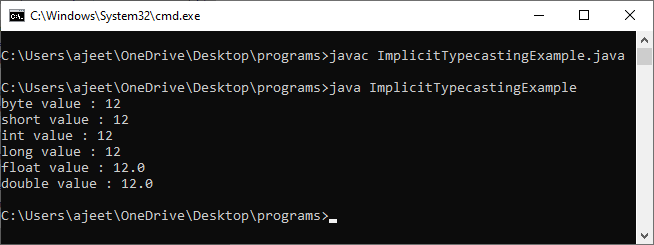
what is implicit casting in java

**The process of converting one type of object and variable into another type** is referred to as Typecasting. When the conversion automatically performs by the compiler without the programmer's interference, it is called implicit type casting or widening casting.

**ImplicitTypecastingExample.java**

1. **public** **class** ImplicitTypecastingExample {
2. **public** **static** **void** main(String args[]) {
3. **byte** p = 12;
4. System.out.println("byte value : "+p);
5. // Implicit Typecasting
6. **short** q = p;
7. System.out.println("short value : "+q);
8. **int** r = q;
9. System.out.println("int value : "+r);
10. **long** s = r;
11. System.out.println("long value : "+s);
12. **float** t = s;
13. System.out.println("float value : "+t);
14. **double** u = t;
15. System.out.println("double value : "+u);
16. }
17. }

**Output:**



**Explanation**

Why should you be careful about String concatenation(+) operator in loops?

If you concatenate Stings in loops for each iteration a new intermediate object is created in the String constant pool. This is not recommended as it causes memory issues. Therefore, concatenating strings in loops as shown in the following example is not recommended.

## **Example**

public class StringExample {

   public static void main(String args[]) {

      String stringArray[] = {"Java", "JavaFX", "HBase", "Oracle"};

      String singleString = new String();

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

         singleString = singleString+stringArray[i]+" ";

      }

      System.out.println(singleString);

   }

}

Can you give examples of different utility methods in String class?

# Empty Checks Methods

## 1. isEmpty(CharSequence cs)

Checks if a CharSequence is empty ("") or null.

public static boolean isEmpty(final CharSequence cs) {

return cs == null || cs.length() == 0;

}

JUnit test case for the above method:

@Test

public void isEmptyTest() {

assertTrue(isEmpty(null));

assertTrue(isEmpty(""));

assertFalse(isEmpty(" "));

assertFalse(isEmpty("bob"));

assertFalse(isEmpty(" bob "));

}

## 2. isNotEmpty(final CharSequence cs)

Checks if a CharSequence is not empty ("") and not null.

public static boolean isNotEmpty(final CharSequence cs) {

return !isEmpty(cs);

}

public static boolean isEmpty(final CharSequence cs) {

return cs == null || cs.length() == 0;

}

Note that isNotEmpty() internally made a call to isEmpty() method.

## 3. isAnyEmpty(final CharSequence... css)

Checks if any of the CharSequences are empty ("") or null.

public static boolean isAnyEmpty(final CharSequence... css) {

if (css != null && css.length == 0) {

return false;

}

for (final CharSequence cs : css){

if (isEmpty(cs)) {

return true;

}

}

return false;

}

Note that isAnyEmpty() internally made a call to isEmpty() method. JUnit test case for isAnyEmpty(final CharSequence... css) method:

@Test

public void isAnyEmptyTest() {

assertTrue(isAnyEmpty((String) null));

assertFalse(isAnyEmpty((String[]) null));

assertTrue(isAnyEmpty(null, "foo"));

assertTrue(isAnyEmpty("", "bar"));

assertTrue(isAnyEmpty("bob", ""));

assertTrue(isAnyEmpty(" bob ", null));

assertFalse(isAnyEmpty("foo", "bar"));

assertFalse(isAnyEmpty(new String[]{}));

assertTrue(isAnyEmpty(new String[]{""}));

}

## 4. isBlank(final CharSequence cs)

Checks if a CharSequence is empty (""), null or whitespace only. This method is similar to isEmpty()but additionally checks for whitespace.

public static boolean isBlank(final CharSequence cs) {

int strLen;

if (cs == null || (strLen = cs.length()) == 0) {

return true;

}

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

if (!Character.isWhitespace(cs.charAt(i))) {

return false;

}

}

return true;

}

JUnit test case for isBlank(final CharSequence cs) method:

@Test

public void isBlankTest() {

assertTrue(isBlank(null));

assertTrue(isBlank(""));

assertTrue(isBlank(" "));

assertFalse(isBlank("bob"));

assertFalse(isBlank(" bob "));

}

## 5. isNotBlank(final CharSequence cs)

Checks if a CharSequence is not empty (""), not null, and not whitespace only.

public static boolean isNotBlank(final CharSequence cs) {

return !isBlank(cs);

}

Note that this method internally made a call to the *isBlank()* method.

## 6. isAnyBlank(final CharSequence... css)

Checks if any of the CharSequences are empty ("") or null or whitespace only.

public static boolean isAnyBlank(final CharSequence... css) {

if (css != null && css.length == 0) {

return false;

}

for (final CharSequence cs : css){

if (isBlank(cs)) {

return true;

}

}

return false;

}

JUnit test case for this method:

@Test

public void isAnyBlankTest() {

assertFalse(isAnyBlank((String[]) null));

assertTrue(isAnyBlank((String) null));

assertTrue(isAnyBlank(null, "foo"));

assertTrue(isAnyBlank(null, null));

assertTrue(isAnyBlank("", "bar"));

assertTrue(isAnyBlank("bob", ""));

assertTrue(isAnyBlank(" bob ", null));

assertTrue(isAnyBlank(" ", "bar"));

assertFalse(isAnyBlank(new String[] {}));

assertTrue(isAnyBlank(new String[] { "" }));

assertFalse(isAnyBlank("foo", "bar"));

}

# Check for Whitespace

## 7. containsWhitespace(CharSequence seq)

Check whether the given CharSequence contains any whitespace characters.

Whitespace is defined by *Character.isWhitespace(char)*.

Returns true if the CharSequence is not empty and contains at least 1 (breaking) whitespace character

public static boolean containsWhitespace(final CharSequence seq) {

if (isEmpty(seq)) {

return false;

}

final int strLen = seq.length();

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

if (Character.isWhitespace(seq.charAt(i))) {

return true;

}

}

return false;

}

JUnit Test Case:

@Test

public void testContainsWhitespace() {

assertFalse(containsWhitespace(null));

assertFalse(containsWhitespace(""));

assertFalse(containsWhitespace("a"));

assertFalse(containsWhitespace("abc"));

assertTrue(containsWhitespace(" "));

assertTrue(containsWhitespace(" a"));

assertTrue(containsWhitespace("abc "));

assertTrue(containsWhitespace("a b"));

assertTrue(containsWhitespace("a b"));

}

# Trim Methods

## 8. trimArrayElements( String[] array)

Trim the elements of the given String array, calling *String.trim()* on each of them.

/\*\*

\* Trim the elements of the given {@code String} array,

\* calling {@code String.trim()} on each of them.

\* @param array the original {@code String} array

\* @return the resulting array (of the same size) with trimmed elements

\*/

public static String[] trimArrayElements( String[] array) {

if (isEmpty(array)) {

return new String[0];

}

String[] result = new String[array.length];

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

String element = array[i];

result[i] = (element != null ? element.trim() : null);

}

return result;

}

## 9. trimWhitespace(String str)

Trim leading and trailing whitespace from the given String.

/\*\*

\* Trim leading and trailing whitespace from the given {@code String}.

\* @param str the {@code String} to check

\* @return the trimmed {@code String}

\* @see java.lang.Character#isWhitespace

\*/

public static String trimWhitespace(String str) {

if (!hasLength(str)) {

return str;

}

int beginIndex = 0;

int endIndex = str.length() - 1;

while (beginIndex <= endIndex && Character.isWhitespace(str.charAt(beginIndex))) {

beginIndex++;

}

while (endIndex > beginIndex && Character.isWhitespace(str.charAt(endIndex))) {

endIndex--;

}

return str.substring(beginIndex, endIndex + 1);

}

JUnit test case:

@Test

public void testTrimWhitespace() {

assertEquals(null, trimWhitespace(null));

assertEquals("", trimWhitespace(""));

assertEquals("", trimWhitespace(" "));

assertEquals("", trimWhitespace("\t"));

assertEquals("a", trimWhitespace(" a"));

assertEquals("a", trimWhitespace("a "));

assertEquals("a", trimWhitespace(" a "));

assertEquals("a b", trimWhitespace(" a b "));

assertEquals("a b c", trimWhitespace(" a b c "));

}

## 10. trimAllWhitespace(String str)

Trim all whitespace from the given String: leading, trailing, and in between characters.

/\*\*

\* Trim <i>all</i> whitespace from the given {@code String}:

\* leading, trailing, and in between characters.

\* @param str the {@code String} to check

\* @return the trimmed {@code String}

\* @see java.lang.Character#isWhitespace

\*/

public static String trimAllWhitespace(String str) {

if (!hasLength(str)) {

return str;

}

int len = str.length();

StringBuilder sb = new StringBuilder(str.length());

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

char c = str.charAt(i);

if (!Character.isWhitespace(c)) {

sb.append(c);

}

}

return sb.toString();

}

JUnit test case:

@Test

public void testTrimAllWhitespace() {

assertEquals("", trimAllWhitespace(""));

assertEquals("", trimAllWhitespace(" "));

assertEquals("", trimAllWhitespace("\t"));

assertEquals("a", trimAllWhitespace(" a"));

assertEquals("a", trimAllWhitespace("a "));

assertEquals("a", trimAllWhitespace(" a "));

assertEquals("ab", trimAllWhitespace(" a b "));

assertEquals("abc", trimAllWhitespace(" a b c "));

}

# Alpha, Numeric, and Case Related Methods

## 11. uppercaseFirstChar(String in)

This method returns the input argument but ensures the first character is capitalized (if possible).

/\*\*

\* Returns the input argument, but ensures the first character is capitalized (if possible).

\* @param in the string to uppercase the first character.

\* @return the input argument, but with the first character capitalized (if possible).

\* @since 1.2

\*/

public static String uppercaseFirstChar(String in) {

if (in == null || in.length() == 0) {

return in;

}

int length = in.length();

StringBuilder sb = new StringBuilder(length);

sb.append(Character.toUpperCase(in.charAt(0)));

if (length > 1) {

String remaining = in.substring(1);

sb.append(remaining);

}

return sb.toString();

}

JUnit test case:

@Test

public void uppercaseFirstCharTest() {

assertEquals("Javaguides", uppercaseFirstChar("javaguides"));

assertEquals("Java", uppercaseFirstChar("java"));

System.out.println(uppercaseFirstChar("javaguides"));

}

## 12. isAlphaSpace(CharSequence cs)

This method checks if the *CharSequence* contains only Unicode letters and space (' ').

public static boolean isAlphaSpace(final CharSequence cs) {

if (cs == null) {

return false;

}

final int sz = cs.length();

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

if (!Character.isLetter(cs.charAt(i)) && cs.charAt(i) != ' ') {

return false;

}

}

return true;

}

JUnit test case:

@Test

public void testIsAlphaspace() {

assertFalse(isAlphaSpace(null));

assertTrue(isAlphaSpace(""));

assertTrue(isAlphaSpace(" "));

assertTrue(isAlphaSpace("a"));

assertTrue(isAlphaSpace("A"));

assertTrue(isAlphaSpace("kgKgKgKgkgkGkjkjlJlOKLgHdGdHgl"));

assertTrue(isAlphaSpace("ham kso"));

assertFalse(isAlphaSpace("1"));

assertFalse(isAlphaSpace("hkHKHik6iUGHKJgU7tUJgKJGI87GIkug"));

assertFalse(isAlphaSpace("\_"));

assertFalse(isAlphaSpace("hkHKHik\*khbkuh"));

}

## 13. isAlphanumericSpace(CharSequence cs)

This method checks if the *CharSequence* contains only Unicode letters, digits, or space (' ').

public static boolean isAlphanumericSpace(final CharSequence cs) {

if (cs == null) {

return false;

}

final int sz = cs.length();

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

if (!Character.isLetterOrDigit(cs.charAt(i)) && cs.charAt(i) != ' ') {

return false;

}

}

return true;

}

JUnit test case:

@Test

public void testIsAlphanumericSpace() {

assertFalse(StringUtility.isAlphanumericSpace(null));

assertTrue(StringUtility.isAlphanumericSpace(""));

assertTrue(StringUtility.isAlphanumericSpace(" "));

assertTrue(StringUtility.isAlphanumericSpace("a"));

assertTrue(StringUtility.isAlphanumericSpace("A"));

assertTrue(StringUtility.isAlphanumericSpace("kgKgKgKgkgkGkjkjlJlOKLgHdGdHgl"));

assertTrue(StringUtility.isAlphanumericSpace("ham kso"));

assertTrue(StringUtility.isAlphanumericSpace("1"));

assertTrue(StringUtility.isAlphanumericSpace("hkHKHik6iUGHKJgU7tUJgKJGI87GIkug"));

assertFalse(StringUtility.isAlphanumericSpace("\_"));

assertFalse(StringUtility.isAlphanumericSpace("hkHKHik\*khbkuh"));

}

## 14. isNumeric(CharSequence cs)

This method checks if the *CharSequence* contains only Unicode digits. A decimal point is not a Unicode digit and returns false.

public static boolean isNumeric(final CharSequence cs) {

if (isEmpty(cs)) {

return false;

}

final int sz = cs.length();

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

if (!Character.isDigit(cs.charAt(i))) {

return false;

}

}

return true;

}

JUnit test case:

@Test

public void testIsNumeric() {

assertFalse(StringUtility.isNumeric(null));

assertFalse(StringUtility.isNumeric(""));

assertFalse(StringUtility.isNumeric(" "));

assertFalse(StringUtility.isNumeric("a"));

assertFalse(StringUtility.isNumeric("A"));

assertFalse(StringUtility.isNumeric("kgKgKgKgkgkGkjkjlJlOKLgHdGdHgl"));

assertFalse(StringUtility.isNumeric("ham kso"));

assertTrue(StringUtility.isNumeric("1"));

assertTrue(StringUtility.isNumeric("1000"));

assertTrue(StringUtility.isNumeric("\u0967\u0968\u0969"));

assertFalse(StringUtility.isNumeric("\u0967\u0968 \u0969"));

assertFalse(StringUtility.isNumeric("2.3"));

assertFalse(StringUtility.isNumeric("10 00"));

assertFalse(StringUtility.isNumeric("hkHKHik6iUGHKJgU7tUJgKJGI87GIkug"));

assertFalse(StringUtility.isNumeric("\_"));

assertFalse(StringUtility.isNumeric("hkHKHik\*khbkuh"));

assertFalse(StringUtility.isNumeric("+123"));

assertFalse(StringUtility.isNumeric("-123"));

}

## 15. isNumericSpace(CharSequence cs)

This method checks if the *CharSequence* contains only Unicode digits or space (' '). A decimal point is not a Unicode digit and returns false.

public static boolean isNumericSpace(final CharSequence cs) {

if (cs == null) {

return false;

}

final int sz = cs.length();

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

if (!Character.isDigit(cs.charAt(i)) && cs.charAt(i) != ' ') {

return false;

}

}

return true;

}

JUnit test case:

@Test

public void testIsNumericSpace() {

assertFalse(StringUtility.isNumericSpace(null));

assertTrue(StringUtility.isNumericSpace(""));

assertTrue(StringUtility.isNumericSpace(" "));

assertFalse(StringUtility.isNumericSpace("a"));

assertFalse(StringUtility.isNumericSpace("A"));

assertFalse(StringUtility.isNumericSpace("kgKgKgKgkgkGkjkjlJlOKLgHdGdHgl"));

assertFalse(StringUtility.isNumericSpace("ham kso"));

assertTrue(StringUtility.isNumericSpace("1"));

assertTrue(StringUtility.isNumericSpace("1000"));

assertFalse(StringUtility.isNumericSpace("2.3"));

assertTrue(StringUtility.isNumericSpace("10 00"));

assertTrue(StringUtility.isNumericSpace("\u0967\u0968\u0969"));

assertTrue(StringUtility.isNumericSpace("\u0967\u0968 \u0969"));

assertFalse(StringUtility.isNumericSpace("hkHKHik6iUGHKJgU7tUJgKJGI87GIkug"));

assertFalse(StringUtility.isNumericSpace("\_"));

assertFalse(StringUtility.isNumericSpace("hkHKHik\*khbkuh"));

}

# File Related String Utility Methods

## 16. fromBufferedReader(BufferedReader bufferedReader)

Converts a *BufferedReader* into a *String*.

public static String fromBufferedReader(BufferedReader bufferedReader) {

StringBuffer sb = new StringBuffer();

try {

String line = bufferedReader.readLine();

while (line != null) {

sb.append(line);

line = bufferedReader.readLine();

if (line != null) {

sb.append("\n");

}

}

} catch (IOException e) {

// replace this with log.error

e.printStackTrace();

}

return sb.toString();

}

## 17. fromInputStream(InputStream inputStream)

Converts an *InputStream* into a *String*.

public static String fromInputStream(InputStream inputStream) {

InputStreamReader inputStreamReader = new InputStreamReader(inputStream);

BufferedReader bufferedReader = new BufferedReader(inputStreamReader);

return fromBufferedReader(bufferedReader);

}

# String Array related Utility Methods

## 18. arrayToMap(String[][] array)

Converts a 2-dimensional array into a map where the first dimension is 2 cell String array containing key and value respectively. Any array with fewer than 2 elements is ignored.

public static Map<String, String> arrayToMap(String[][] array) {

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

for (String[] pair : array) {

if (pair.length > 1) {

// got a pair, add to map

map.put(pair[0], pair[1]);

}

}

return map;

}

## 19. startsWith(String[] array, String startsWith)

Searches through a given String array and returns an element that starts with the supplied startsWith string. This method ignores the case. If no match can be found then an empty string is returned.

public static String startsWith(String[] array, String startsWith) {

String lcStartsWith = startsWith.toLowerCase();

for (String element : array) {

if (element.toLowerCase().startsWith(lcStartsWith)) {

return element;

}

}

return "";

}

# Character related Utility Methods

## 20. trimLeadingCharacter(String str, char leadingCharacter)

This method trims all occurrences of the supplied leading character from the given String.

/\*\*

\* Trim all occurrences of the supplied leading character from the given {@code String}.

\* @param str the {@code String} to check

\* @param leadingCharacter the leading character to be trimmed

\* @return the trimmed {@code String}

\*/

public static String trimLeadingCharacter(String str, char leadingCharacter) {

if (!hasLength(str)) {

return str;

}

StringBuilder sb = new StringBuilder(str);

while (sb.length() > 0 && sb.charAt(0) == leadingCharacter) {

sb.deleteCharAt(0);

}

return sb.toString();

}

JUnit test case:

@Test

public void testTrimLeadingCharacter() {

assertEquals(null, StringUtility.trimLeadingCharacter(null, ' '));

assertEquals("", StringUtility.trimLeadingCharacter("", ' '));

assertEquals("", StringUtility.trimLeadingCharacter(" ", ' '));

assertEquals("\t", StringUtility.trimLeadingCharacter("\t", ' '));

assertEquals("a", StringUtility.trimLeadingCharacter(" a", ' '));

assertEquals("a ", StringUtility.trimLeadingCharacter("a ", ' '));

assertEquals("a ", StringUtility.trimLeadingCharacter(" a ", ' '));

assertEquals("a b ", StringUtility.trimLeadingCharacter(" a b ", ' '));

assertEquals("a b c ", StringUtility.trimLeadingCharacter(" a b c ", ' '));

}

## 21. trimTrailingCharacter(String str, char trailingCharacter)

This method trims all occurrences of the supplied trailing character from the given String.

/\*\*

\* Trim all occurrences of the supplied trailing character from the given {@code String}.

\* @param str the {@code String} to check

\* @param trailingCharacter the trailing character to be trimmed

\* @return the trimmed {@code String}

\*/

public static String trimTrailingCharacter(String str, char trailingCharacter) {

if (!hasLength(str)) {

return str;

}

StringBuilder sb = new StringBuilder(str);

while (sb.length() > 0 && sb.charAt(sb.length() - 1) == trailingCharacter) {

sb.deleteCharAt(sb.length() - 1);

}

return sb.toString();

}

JUnit test case:

@Test

public void testTrimTrailingCharacter() {

assertEquals(null, StringUtility.trimTrailingCharacter(null, ' '));

assertEquals("", StringUtility.trimTrailingCharacter("", ' '));

assertEquals("", StringUtility.trimTrailingCharacter(" ", ' '));

assertEquals("\t", StringUtility.trimTrailingCharacter("\t", ' '));

assertEquals("a", StringUtility.trimTrailingCharacter("a ", ' '));

assertEquals(" a", StringUtility.trimTrailingCharacter(" a", ' '));

assertEquals(" a", StringUtility.trimTrailingCharacter(" a ", ' '));

assertEquals(" a b", StringUtility.trimTrailingCharacter(" a b ", ' '));

assertEquals(" a b c", StringUtility.trimTrailingCharacter(" a b c ", ' '));

}

# Other Utility Methods

## 22. startsWithIgnoreCase( String str, String prefix)

Test if the given string starts with the specified prefix, ignoring upper/lower case.

/\*\*

\* Test if the given {@code String} starts with the specified prefix,

\* ignoring upper/lower case.

\* @param str the {@code String} to check

\* @param prefix the prefix to look for

\* @see java.lang.String#startsWith

\*/

public static boolean startsWithIgnoreCase( String str, String prefix) {

return (str != null && prefix != null && str.length() >= prefix.length() &&

str.regionMatches(true, 0, prefix, 0, prefix.length()));

}

## 23. quote( String str)

Quote the given String with single quotes.

/\*\*

\* Quote the given {@code String} with single quotes.

\* @param str the input {@code String} (e.g. "myString")

\* @return the quoted {@code String} (e.g. "'myString'"),

\* or {@code null} if the input was {@code null}

\*/

public static String quote( String str) {

return (str != null ? "'" + str + "'" : null);

}

## 24. quoteIfString( Object obj)

Turn the given Object into a String with single quotes if it is a String; keeping the Object as-is else.

/\*\*

\* Turn the given Object into a {@code String} with single quotes

\* if it is a {@code String}; keeping the Object as-is else.

\* @param obj the input Object (e.g. "myString")

\* @return the quoted {@code String} (e.g. "'myString'"),

\* or the input object as-is if not a {@code String}

\*/

public static Object quoteIfString( Object obj) {

return (obj instanceof String ? quote((String) obj) : obj);

}

## 25. mergeStringArrays(String array1[], String array2[])

/\*\*

\* This String utility or util method can be used to merge 2 arrays of

\* string values. If the input arrays are like this array1 = {"a", "b" ,

\* "c"} array2 = {"c", "d", "e"} Then the output array will have {"a", "b" ,

\* "c", "d", "e"}

\*

\* This takes care of eliminating duplicates and checks null values.

\*

\* @param values

\* @return

\*/

public static String[] mergeStringArrays(String array1[], String array2[]) {

if (array1 == null || array1.length == 0)

return array2;

if (array2 == null || array2.length == 0)

return array1;

List<String> array1List = Arrays.asList(array1);

List<String> array2List = Arrays.asList(array2);

List<String> result = new ArrayList<String>(array1List);

List<String> tmp = new ArrayList<String>(array1List);

tmp.retainAll(array2List);

result.removeAll(tmp);

result.addAll(array2List);

return ((String[]) result.toArray(new String[result.size()]));

}

public static void main(String[] args) {

String[] strArray = mergeStringArrays(new String[] {"abc","xyz","pqr"}, new String[] {"ABC","PQR"});

for(String string : strArray){

System.out.println(string);

}

}

Output:

abc

xyz

pqr

ABC

PQR

## 26. endsWithIgnoreCase(String str, String suffix)

Check a string ends with another string ignoring the case.

/\*\*

\* Check a String ends with another string ignoring the case.

\*

\* @param str

\* @param suffix

\* @return

\*/

public static boolean endsWithIgnoreCase(String str, String suffix) {

if (str == null || suffix == null) {

return false;

}

if (str.endsWith(suffix)) {

return true;

}

if (str.length() < suffix.length()) {

return false;

} else {

return str.toLowerCase().endsWith(suffix.toLowerCase());

}

}

## 27. startsWithIgnoreCase(String str, String prefix)

Check a string starts with another string ignoring the case.

/\*\*

\* Check a String starts with another string ignoring the case.

\*

\* @param str

\* @param prefix

\* @return

\*/

public static boolean startsWithIgnoreCase(String str, String prefix) {

if (str == null || prefix == null) {

return false;

}

if (str.startsWith(prefix)) {

return true;

}

if (str.length() < prefix.length()) {

return false;

} else {

return str.toLowerCase().startsWith(prefix.toLowerCase());

}

}

What is a class?

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

* **Fields**
* **Methods**
* **Constructors**
* **Blocks**
* **Nested class and interface**

Syntax to declare a class:

1. **class** <class\_name>{
2. field;
3. method;
4. }

What is an object?

What is an object in Java

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

Skip Ad

* **State:** represents the data (value) of an object.
* **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
* **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

**An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

**Object Definitions:**

* An object is *a real-world entity*.
* An object is *a runtime entity*.
* The object is *an entity which has state and behavior*.
* The object is *an instance of a class*.

What is state of an object?

State of an object - The state or attributes are **the built in characteristics or properties of an object**. For example, a T.V has the size, colour, model etc. Behaviour of the object - The behavior or operations of an object are its predefined functions. For example, a T.V.

What is the super class of every class in Java?

The class named Object is the super class of every class in Java.

Explain about toString method ?

## **Description**

The method is used to get a String object representing the value of the Number Object.

If the method takes a primitive data type as an argument, then the String object representing the primitive data type value is returned.

If the method takes two arguments, then a String representation of the first argument in the radix specified by the second argument will be returned.

## **Syntax**

Following are all the variants of this method −

String toString()

static String toString(int i)

## **Parameters**

Here is the detail of parameters −

* **i** − An int for which string representation would be returned.

## **Return Value**

* **toString()** − This returns a String object representing the value of **this** Integer.
* **toString(int i)** − This returns a String object representing the specified integer.

## **Example**

[Live Demo](http://tpcg.io/QV9fam)

public class Test {

public static void main(String args[]) {

Integer x = 5;

System.out.println(x.toString());

System.out.println(Integer.toString(12));

}

}

This will produce the following result −

## **Output**

5

12

What is the use of equals method in Java?

The equals() method compares two strings, and returns true if the strings are equal, and false if not.

 What are the important things to consider when implementing equals method?

**The equals() method must be:**

* reflexive: an object must equal itself.
* symmetric: x. equals(y) must return the same result as y. equals(x)
* transitive: if x. equals(y) and y. ...
* consistent: the value of equals() should change only if a property that is contained in equals() changes (no randomness allowed)

What is the Hashcode method used for in Java?

hashCode in Java is a function that **returns the hashcode value of an object on calling**. It returns an integer or a 4 bytes value which is generated by the hashing algorithm. The process of assigning a unique value to an object or attribute using an algorithm, which enables quicker access, is known as hashing.24

## What is Inheritance?

**Inheritance** is a mechanism in which one class acquires the property of another class. For example, a child inherits the traits of his/her parents. With inheritance, we can reuse the fields and methods of the existing class. Hence, inheritance facilitates Reusability and is an important concept of OOPs.

In this tutorial, you will learn-

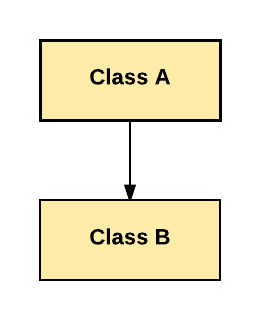
* [Types of Inheritance](https://www.guru99.com/java-class-inheritance.html#types-of-inheritance)
* [Inheritance in Java](https://www.guru99.com/java-class-inheritance.html#inheritance-in-java)
* [Java Inheritance Example](https://www.guru99.com/java-class-inheritance.html#java-inheritance-example)
* [Super Keyword](https://www.guru99.com/java-class-inheritance.html#super-keyword)
* [Learn Inheritance in OOP’s with Example](https://www.guru99.com/java-class-inheritance.html#learn-inheritance-in-oops-with-example)

## Types of Inheritance

Here are the different types of inheritance in Java:

### Single Inheritance:

In Single Inheritance one class extends another class (one class only).

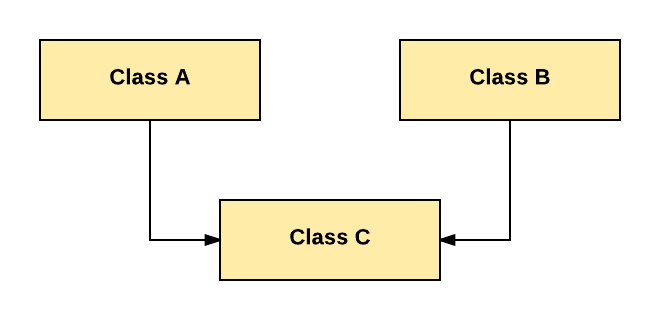
Single Inheritance

In above diagram, Class B extends only Class A. Class A is a super class and Class B is a Sub-class.

EXPLORE MORELearn Java Programming with Beginners Tutorial08:32Linux Tutorial for Beginners: Introduction to Linux Operating...01:35What is Integration Testing Software Testing Tutorial03:04What is JVM (Java Virtual Machine) with Architecture JAVA...02:24How to write a TEST CASE Software Testing Tutorial01:08Seven Testing Principles Software Testing05:01Linux File Permissions Commands with Examples13:29How to use Text tool in Photoshop CC Tutorial08:32What is NoSQL Database Tutorial02:00Important Linux Commands for Beginners Linux Tutorial15:03

### Multiple Inheritance:

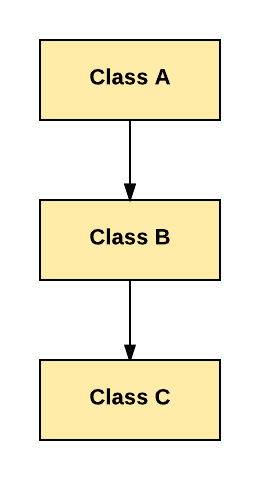
Multiple Inheritance is one of the inheritance in Java types where one class extending more than one class. Java does not support multiple inheritance.

Java Multiple Inheritance

As per above diagram, Class C extends Class A and Class B both.

### Multilevel Inheritance:

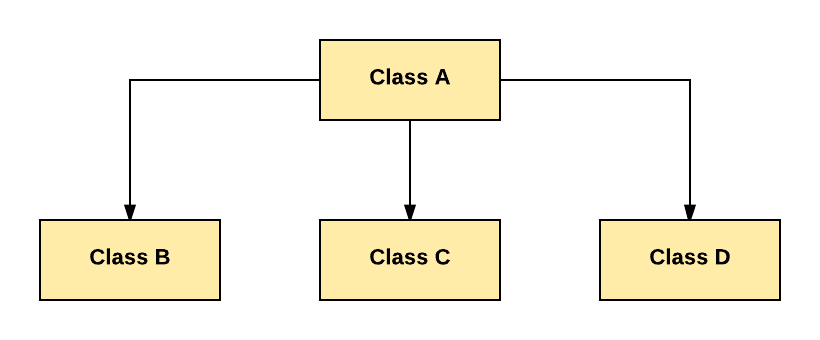
In Multilevel Inheritance, one class can inherit from a derived class. Hence, the derived class becomes the base class for the new class.

Multilevel Inheritance

As per shown in diagram Class C is subclass of B and B is a of subclass Class A.

### Hierarchical Inheritance:

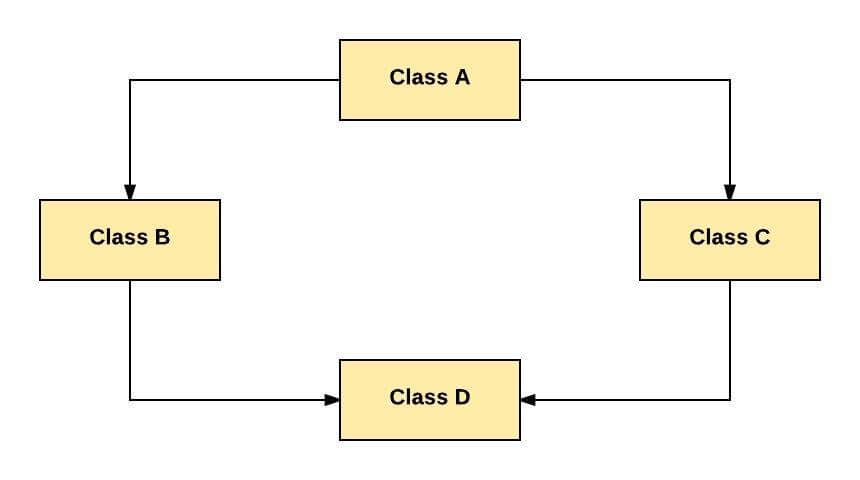
In Hierarchical Inheritance, one class is inherited by many sub classes.

Hierarchical Inheritance

As per above example, Class B, C, and D inherit the same class A.

### Hybrid Inheritance:

Hybrid inheritance is one of the inheritance types in Java which is a combination of Single and Multiple inheritance.

Hybrid Inheritance in Java

As per above example, all the public and protected members of Class A are inherited into Class D, first via Class B and secondly via Class C.

**Note:** Java doesn’t support hybrid/Multiple inheritence

## Inheritance In Java

**Java Inheritance** is a mechanism in which one class acquires the property of another class. In Java, when an “Is-A” relationship exists between two classes, we use Inheritance. The parent class is called a super class and the inherited class is called a subclass. The keyword extends is used by the sub class to inherit the features of super class.

Inheritance is important since it leads to the reusability of code.

**Java Inheritance Syntax:**

class subClass extends superClass

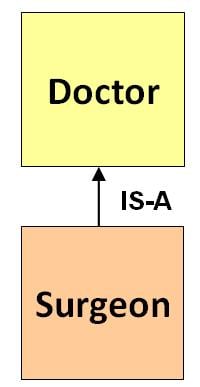
{

//methods and fields

}

## Java Inheritance Example

Here is an example of inheritance in Java:



Java Inheritance Example

class Doctor {

void Doctor\_Details() {

System.out.println("Doctor Details...");

}

}

class Surgeon extends Doctor {

void Surgeon\_Details() {

System.out.println("Surgen Detail...");

}

}

public class Hospital {

public static void main(String args[]) {

Surgeon s = new Surgeon();

s.Doctor\_Details();

s.Surgeon\_Details();

}

}

What is method overloading?

# **Method Overloading in Java**

1. [Different ways to overload the method](https://www.javatpoint.com/method-overloading-in-java#monumberofways)
2. [By changing the no. of arguments](https://www.javatpoint.com/method-overloading-in-java#mobynumber)
3. [By changing the datatype](https://www.javatpoint.com/method-overloading-in-java#mobydatatype)
4. [Why method overloading is not possible by changing the return type](https://www.javatpoint.com/method-overloading-in-java#moreturntype)
5. [Can we overload the main method](https://www.javatpoint.com/method-overloading-in-java#momainmethod)
6. [method overloading with Type Promotion](https://www.javatpoint.com/method-overloading-in-java#motypepromotion)

If a [class](https://www.javatpoint.com/object-and-class-in-java) has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the [program](https://www.javatpoint.com/java-programs).

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

3.4M

Xbox Game Pass To Reportedly Offer Family Plan Soon

## **Advantage of method overloading**

Method overloading increases the readability of the program.

### Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

#### **In Java, Method Overloading is not possible by changing the return type of the method only.**

### 1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating [static methods](https://www.javatpoint.com/static-keyword-in-java) so that we don't need to create instance for calling methods.

1. **class** Adder{
2. **static** **int** add(**int** a,**int** b){**return** a+b;}
3. **static** **int** add(**int** a,**int** b,**int** c){**return** a+b+c;}
4. }
5. **class** TestOverloading1{
6. **public** **static** **void** main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(11,11,11));
9. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestOverloading1)

Output:

22

33

What is method overriding?

# **Method Overriding in Java**

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

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

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

### Usage of Java Method Overriding

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

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

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



### Understanding the problem without method overriding

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

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

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

Output:

Vehicle is running

 Can super class reference variable can hold an object of sub class?

Yes, the super class reference variable can hold the sub class object actually, it is widening in case of objects (Conversion of lower datatype to a higher datatype).

But, using this reference you can access the members of super class only, if you try to access the sub class members a compile time error will be generated.

## **Example**

In the following Java example, we have two classes namely Person and Student. The Person class has two instance variables name and age and one instance method displayPerson() which displays the name and age.

The Student extends the person class and in addition to the inherited name and age it has two more variables branch and student\_id. It has a method displayData() which displays all four values.

In the main method, we are assigning the subclass object with the super class reference variable

class Person{

   private String name;

   private int age;

   public Person(String name, int age){

      this.name = name;

      this.age = age;

   }

   public void displayPerson() {

      System.out.println("Data of the Person class: ");

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

      System.out.println("Age: "+this.age);

   }

}

public class Student extends Person {

   public String branch;

   public int Student\_id;

   public Student(String name, int age, String branch, int Student\_id){

      super(name, age);

      this.branch = branch;

      this.Student\_id = Student\_id;

   }

   public void displayStudent() {

      System.out.println("Data of the Student class: ");

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

      System.out.println("Age: "+this.age);

      System.out.println("Branch: "+this.branch);

      System.out.println("Student ID: "+this.Student\_id);

   }

   public static void main(String[] args) {

      Person person = new Student("Krishna", 20, "IT", 1256);

      person.displayPerson();

   }

}

## **Output**

Data of the Person class:

Name: Krishna

Age: 20

Is multiple inheritance allowed in Java?

**The Java programming language supports multiple inheritance of type**, which is the ability of a class to implement more than one interface. An object can have multiple types: the type of its own class and the types of all the interfaces that the class implements.

What is an interface?

# **Interface in Java**

1. [Interface](https://www.javatpoint.com/interface-in-java)
2. [Example of Interface](https://www.javatpoint.com/interface-in-java#interfaceex)
3. [Multiple inheritance by Interface](https://www.javatpoint.com/interface-in-java#interfacemultiple)
4. [Why multiple inheritance is supported in Interface while it is not supported in case of class.](https://www.javatpoint.com/interface-in-java#interfacewhynot)
5. [Marker Interface](https://www.javatpoint.com/interface-in-java#interfacemarker)
6. [Nested Interface](https://www.javatpoint.com/nested-interface)

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a mechanism to achieve [*abstraction*](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/inheritance-in-java).

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**.

Competitive questions on Structures in HindiKeep Watching

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface.

Since Java 9, we can have **private methods** in an interface.

## **Why use Java interface?**

There are mainly three reasons to use interface. They are given below.

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.



## **How to declare an interface?**

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

### Syntax:

1. **interface** <interface\_name>{
3. // declare constant fields
4. // declare methods that abstract
5. // by default.
6. }

## **Java 8 Interface Improvement**

Since [Java 8](https://www.javatpoint.com/java-8-features), interface can have default and static methods which is discussed later.

## **Internal addition by the compiler**

#### **The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.**

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.



#### **The relationship between classes and interfaces**

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



## **Java Interface Example**

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

1. **interface** printable{
2. **void** print();
3. }
4. **class** A6 **implements** printable{
5. **public** **void** print(){System.out.println("Hello");}
7. **public** **static** **void** main(String args[]){
8. A6 obj = **new** A6();
9. obj.print();
10. }
11. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=A6)

Output:

Hello

Can you explain a few tricky things about interfaces?

* An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
* An interface is not extended by a class; it is implemented by a class.
* An interface can extend multiple interfaces.

Can you extend an interface?

An interface contains variables and methods like a class but the methods in an interface are abstract by default unlike a class. An interface extends another interface like a class implements an interface in interface inheritance.

A program that demonstrates extending interfaces in Java is given as follows:

## **Example**

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

interface A {

   void funcA();

}

interface B extends A {

   void funcB();

}

class C implements B {

   public void funcA() {

      System.out.println("This is funcA");

   }

   public void funcB() {

      System.out.println("This is funcB");

   }

}

public class Demo {

   public static void main(String args[]) {

      C obj = new C();

      obj.funcA();

      obj.funcB();

   }

}

## **Output**

This is funcA

This is funcB

 Can a class extend multiple interfaces?

# **Can an interface extend multiple interfaces in Java?**

Yes, we can do it. An interface can extend multiple interfaces in Java.

## **Example:**

interface A {

   public void test();

   public void test1();

}

interface B {

   public void test();

   public void test2();

}

interface C extends A,B {

   public void test3();

}

class D implements C {

   public void test() {

      System.out.println("Testing\n");

   }

   public void test1() {

      System.out.println("Testing1\n");

   }

   public void test2() {

      System.out.println("Testing2\n");

   }

   public void test3() {

      System.out.println("Testing3");

   }

}

public class Main {

   public static void main(String[] args) {

      D d=new D();

      d.test();

      d.test1();

      d.test2();

      d.test3();

   }

 }

## **Output:**

Testing

Testing1

Testing2

Testing3

Compare abstract class vs interface?

# **Difference between abstract class and interface**

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

### Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

54.5M

1K

Difference between JDK, JRE, and JVM

1. //Creating interface that has 4 methods
2. **interface** A{
3. **void** a();//bydefault, public and abstract
4. **void** b();
5. **void** c();
6. **void** d();
7. }
9. //Creating abstract class that provides the implementation of one method of A interface
10. **abstract** **class** B **implements** A{
11. **public** **void** c(){System.out.println("I am C");}
12. }
14. //Creating subclass of abstract class, now we need to provide the implementation of rest of the methods
15. **class** M **extends** B{
16. **public** **void** a(){System.out.println("I am a");}
17. **public** **void** b(){System.out.println("I am b");}
18. **public** **void** d(){System.out.println("I am d");}
19. }
21. //Creating a test class that calls the methods of A interface
22. **class** Test5{
23. **public** **static** **void** main(String args[]){
24. A a=**new** M();
25. a.a();
26. a.b();
27. a.c();
28. a.d();
29. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Test5)

Output:

I am a

I am b

I am c

I am d

What is constructor

## **What Does Constructor Mean?**

A constructor is a special method of a class or structure in object-oriented programming that initializes a newly created object of that type. Whenever an object is created, the constructor is called automatically.

How do you call a super class constructor from a constructor?

## 3. Use of super() to access superclass constructor

As we know, when an object of a class is created, its default constructor is automatically called.

To explicitly call the superclass constructor from the subclass constructor, we use super(). It's a special form of the super keyword.

super() can be used only inside the subclass constructor and must be the first statement.

### Example 4: Use of super()

class Animal {

// default or no-arg constructor of class Animal

Animal() {

System.out.println("I am an animal");

}

}

class Dog extends Animal {

// default or no-arg constructor of class Dog

Dog() {

// calling default constructor of the superclass

super();

System.out.println("I am a dog");

}

}

class Main {

public static void main(String[] args) {

Dog dog1 = new Dog();

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

I am an animal

I am a dog

Can a constructor be called directly from a method?

## **nvoking a constructor from a method**

No, you cannot call a constructor from a method. The only place from which you can invoke constructors using “this()” or, “super()” is the first line of another constructor. If you try to invoke constructors explicitly elsewhere, a compile time error will be generated.

### Example

import java.util.Scanner;

public class Student {

   private String name;

   private int age;

   Student(){}

   Student(String name, int age){

      this.name = name;

      this.age = age;

   }

   public void SetValues(String name, int age){

      this(name, age);

   }

   public void display() {

      System.out.println("name: "+this.name);

      System.out.println("age: "+this.age);

   }

   public static void main(String args[]) {

      Scanner sc = new Scanner(System.in);

      System.out.println("Enter the name of the student: ");

      String name = sc.nextLine();

      System.out.println("Enter the age of the student: ");

      int age = sc.nextInt();

      Student obj = new Student();

      obj.SetValues(name, age);

      obj.display();

   }

}

### Compile time error

Student.java:12: error: call to this must be first statement in constructor

   this(name, age);

^

1 error

 Is a super class constructor called even when there is no explicit call from a sub class constructor?

**subclass implicitly call even default constructor present in super class which is non parameterised**. We have to call explicitly when we pass parameters to the constructor

What is the use of instanceof operator in Java?

The instanceof operator in Java is used **to check whether an object is an instance of a particular class or not**. objectName instanceOf className; Here, if objectName is an instance of className , the operator returns true . Otherwise, it returns false

The instanceof operator in Java is used to check whether an object is an instance of a particular class or not.

Its syntax is

objectName instanceOf className;

Here, if objectName is an instance of className, the operator returns true. Otherwise, it returns false.

### Example: Java instanceof

class Main {

public static void main(String[] args) {

// create a variable of string type

String name = "Programiz";

// checks if name is instance of String

boolean result1 = name instanceof String;

System.out.println("name is an instance of String: " + result1);

// create an object of Main

Main obj = new Main();

// checks if obj is an instance of Main

boolean result2 = obj instanceof Main;

System.out.println("obj is an instance of Main: " + result2);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

name is an instance of String: true

obj is an instance of Main: true

## What is Coupling in Java?

Coupling is nothing but the dependency of one class on the other. If one object in a code uses the other object in the program, it is called **loose coupling in Java**. In coupling, two classes or objects collaborate and work with each other to complete a pre-defined task. It simply means that one element requires another element to complete a function. It is known as collaboration when one class calls the logic of the other class.

## What are the Types of Coupling?

Coupling in Java is of two types.

### Loose Coupling in Java

When two classes, modules, or components have low dependencies on each other, it is called**loose coupling in Java**. **Loose coupling in Java** means that the classes are independent of each other. The only knowledge one class has about the other class is what the other class has exposed through its interfaces in loose coupling. If a situation requires objects to be used from outside, it is termed as a **loose coupling** situation.

Here, the parent object is rarely using the object, and the object can be easily changed from external sources. The **loose coupling in Java** has the edge over tight coupling as it reduces code maintenance and efforts. A change in one class does not require changes in the other class, and two classes can work independently.

**Example 1:** Imagine you have created two classes, A and B, in your program. Class A is called volume, and class B evaluates the volume of a cylinder. If you change class A volume, then you are not forced to change class B. This is called **loose coupling in Java**. When class A requires changes in class B, then you have tight coupling.

**Code**

|  |
| --- |
| package loosecoupling;   class Volume {     public static void main(String args[]) {          Cylinder b = new Cylinder(25, 25, 25);             System.out.println(b.getVolume());     }  }  final class Cylinder {      private int volume;      Cylinder(int length, int width, int height) {               this.volume = length \* width \* height;      }      public int getVolume() {               return volume;      }  } |

**Explanation:** In the above example, class A and class B are loosely coupled.

## Differences Between Loose Coupling and Tight Coupling

The following table lists the differences between**loose coupling** and tight coupling.

|  |  |
| --- | --- |
| **Loose Coupling** | **Tight Coupling** |
| Objects are independent of each other. | One object is dependent on the other object to complete a task. |
| Better testability | Testability is not as great as the **loose coupling in Java**. |
| Asynchronous communication | Synchronous communication |
| Less coordination. Swapping code between two classes is not easy. | Provides better coordination. You can easily swap code between two objects. |
| No concept of interface | Follows GOF principles to interface |
| Less information flow | More information flow |
| Highly changeable | It does not have the change capability. |

What is cohesion?

# Cohesion in Java

* **Difficulty Level :** [Basic](https://www.geeksforgeeks.org/basic/)
* **Last Updated :** 31 May, 2022

Cohesion in Java is the Object-Oriented principle most closely associated with making sure that a class is designed with a single, well-focused purpose. In object-oriented design, cohesion refers all to how a single class is designed.

***Note:****The more focused a class is, the more is the cohesiveness of that class.*

The advantage of high cohesion is that such classes are much easier to maintain (and less frequently changed) than classes with low cohesion. Another benefit of high cohesion is that classes with a well-focused purpose tend to be more reusable than other classes.

**Example:**Suppose we have a class that multiplies two numbers, but the same class creates a pop-up window displaying the result. This is an example of a low cohesive class because the window and the multiplication operation don’t have much in common. To make it high cohesive, we would have to create a class Display and a class Multiply. The Display will call Multiply’s method to get the result and display it. This way to develop a high cohesive solution.

**Let us understand the structure of the** **high cohesive program:**

* Java

|  |
| --- |
| // Java program to illustrate  // high cohesive behavior    class Multiply {        int a = 5;      int b = 5;        public int mul(int a, int b)      {          this.a = a;          this.b = b;          return a \* b;      }  }    class Display {      public static void main(String[] args)      {          Multiply m = new Multiply();          System.out.println(m.mul(5, 5));      }  } |

**Output**

25

 Can you create an inner class inside a method?

## **Method-local Inner Class**

In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.

A method-local inner class can be instantiated only within the method where the inner class is defined. The following program shows how to use a method-local inner class.

**Example**

[Live Demo](http://tpcg.io/v5bNDt)

public class Outerclass {

// instance method of the outer class

void my\_Method() {

int num = 23;

// method-local inner class

class MethodInner\_Demo {

public void print() {

System.out.println("This is method inner class "+num);

}

} // end of inner class

// Accessing the inner class

MethodInner\_Demo inner = new MethodInner\_Demo();

inner.print();

}

public static void main(String args[]) {

Outerclass outer = new Outerclass();

outer.my\_Method();

}

}

If you compile and execute the above program, you will get the following result −

**Output**

This is method inner class 23

What is an anonymous class?

A nested class that doesn't have any name is known as an anonymous class.

An anonymous class must be defined inside another class. Hence, it is also known as an anonymous inner class. Its syntax is:

class outerClass {

// defining anonymous class

object1 = new Type(parameterList) {

// body of the anonymous class

};

}

Anonymous classes usually extend subclasses or implement interfaces.

Here, **Type** can be

1. a superclass that an anonymous class extends
2. an interface that an anonymous class implements

The above code creates an object, object1, of an anonymous class at runtime.

**Note:** Anonymous classes are defined inside an expression. So, the semicolon is used at the end of anonymous classes to indicate the end of the expression.

**Example 1: Anonymous Class Extending a Class**

class Polygon {

public void display() {

System.out.println("Inside the Polygon class");

}

}

class AnonymousDemo {

public void createClass() {

// creation of anonymous class extending class Polygon

Polygon p1 = new Polygon() {

public void display() {

System.out.println("Inside an anonymous class.");

}

};

p1.display();

}

}

class Main {

public static void main(String[] args) {

AnonymousDemo an = new AnonymousDemo();

an.createClass();

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

Inside an anonymous class.

What access types of variables can be accessed from a class in same package?

**Protected Access Modifier** - Protected

Variables, methods, and constructors, which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

What access types of variables can be accessed from a class in different package?

An access level determines whether other classes can use a particular member variable or call a particular method. The Java programming language supports four access specifiers for member variables and methods: **private , protected , public , and, if left unspecified, package private**.

What happens when a variable is marked as volatile?

When we declare a variable volatile, it means that **all reads and all writes to this variable or from this variable will go directly into the main memory**. The values of these variables will never be cached.

Why should you always use blocks around if statement?

The Only Reason To Use Curly Brackets Blocks For IF-Statements. **Without curly brackets, you could accidentally write a semicolon after the IF-statements**. The semicolon is a valid, empty statement, which will be "execute" instead of the actual (intended) one.

Why is exception handling important?

ava exception handling is important because **it helps maintain the normal, desired flow of the program even when unexpected events occur**. If Java exceptions are not handled, programs may crash or requests may fail. This can be very frustrating for customers and if it happens repeatedly, you could lose those customers.

What design pattern is used to implement exception handling features in most languages?

We can view **Chain Of Responsibility** — Design Pattern for this purpose. Example: When an exception is thrown in a method, the runtime checks to see if the method has a mechanism to handle the exception or if it should be passed up the call stack.

**A checked exception is caught at compile time whereas a runtime or unchecked exception is, as it states, at runtime**. A checked exception must be handled either by re-throwing or with a try catch block, whereas an unchecked isn't required to be handled.

 What happens when you throw a checked exception from a method?

Checked exceptions are checked at compile-time. It means if a method is throwing a checked exception then it should handle the exception using try-catch block or it should declare the exception using throws keyword, otherwise **the program will give a compilation error**.

How do you create a custom exception?

## **Steps to create a Custom Exception with an Example**

* CustomException class is the custom exception class this class is extending Exception class.
* Create one local variable message to store the exception message locally in the class object.
* We are passing a string argument to the constructor of the custom exception object. The constructor set the argument string to the private string message.
* toString() method is used to print out the exception message.
* We are simply throwing a CustomException using one try-catch block in the main method and observe how the string is passed while creating a custom exception. Inside the catch block, we are printing out the message.

## **Example**

class CustomException extends Exception {

   String message;

   CustomException(String str) {

      message = str;

   }

   public String toString() {

      return ("Custom Exception Occurred : " + message);

   }

}

public class MainException {

   public static void main(String args[]) {

      try {

         throw new CustomException("This is a custom message");

      } catch(CustomException e) {

         System.out.println(e);

      }

   }

}

How do you handle multiple exception types with same exception handling block?

What Java 7 provides us:

1. **public** **class** MultipleExceptionExample{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. **int** array[] = newint[10];
5. array[10] = 30/0;
6. }
7. **catch**(ArithmeticException | ArrayIndexOutOfBoundsException e){
8. System.out.println(e.getMessage());
9. }
10. }
11. }

Can you explain about try with resources?

The try -with-resources statement is **a try statement that declares one or more resources**. A resource is an object that must be closed after the program is finished with it. The try -with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.

static String readFirstLineFromFileWithFinallyBlock(String path) throws IOException {

FileReader fr = new FileReader(path);

BufferedReader br = new BufferedReader(fr);

try {

return br.readLine();

} finally {

br.close();

fr.close();

}

}

What are the default values in an array?

|  |  |  |
| --- | --- | --- |
| 1 | boolean | false |
| 2 | int | 0 |
| 3 | double | 0.0 |
| 4 | String | null |
| 5 | User-Defined Type | null |

How do you loop around an array using enhanced for loop?

There is a special kind of loop that can be used with arrays that is called an **enhanced for loop** or a **for each loop**. This loop is much easier to write because it does not involve an index variable or the use of the []. It just sets up a variable that is set to each value in the array successively.

To set up a for-each loop, use **for (type variable : arrayname)** where the type is the type for elements in the array, and read it as “for each variable value in arrayname”. You may have used a similar loop in AP CSP Pseudocode or App Inventor with lists like below.

How do you print the content of an array?

# ava Program to Print an Array

In this program, you'll learn different techniques to print the elements of a given array in Java.

To understand this example, you should have the knowledge of the following [Java programming](https://www.programiz.com/java-programming) topics:

* [Java Arrays](https://www.programiz.com/java-programming/arrays)
* [Java Multidimensional Arrays](https://www.programiz.com/java-programming/multidimensional-array)
* [Java for Loop](https://www.programiz.com/java-programming/for-loop)

## Example 1: Print an Array using For loop

public class Array {

public static void main(String[] args) {

int[] array = {1, 2, 3, 4, 5};

for (int element: array) {

System.out.println(element);

}

}

}

**Output**

1

2

3

4

5

In the above program, the for-each loop is used to iterate over the given array, array.

It accesses each element in the array and prints using println().

## Example 2: Print an Array using standard library Arrays

import java.util.Arrays;

public class Array {

public static void main(String[] args) {

int[] array = {1, 2, 3, 4, 5};

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

}

}

**Output**

[1, 2, 3, 4, 5]

In the above program, the for loop has been replaced by a single line of code using Arrays.toString() function.

As you can see, this gives a clean output without any extra lines of code.

## Example 3: Print a Multi-dimensional Array

import java.util.Arrays;

public class Array {

public static void main(String[] args) {

int[][] array = {{1, 2}, {3, 4}, {5, 6, 7}};

System.out.println(Arrays.deepToString(array));

}

}

**Output**

[[1, 2], [3, 4], [5, 6, 7]]

How do you compare two arrays?

# Compare Two Arrays in Java

* **Difficulty Level :** [Medium](https://www.geeksforgeeks.org/medium/)
* **Last Updated :** 31 May, 2022

While comparing two arrays we can not use “==” operator as it will compare the addresses of the memory block to which both the arrays are pointing. If we want to compare the elements inside the array we need to figure out other ways instead of using arithmetic operators. As we all know arrays data structures possess the property of containing elements in a continuous manner because of which we can calculate the size of both the arrays using the size() method of Arrays class itself, and can start comparing the indices if the size of both arrays is found to be equal.

**Illustration:**

* Java

|  |
| --- |
| // Java Program to Illustrate  // Comparison of Arrays  // Using == Operators    // Importing required classes  import java.util.\*;    // Main class  class GFG {        // Main driver method      public static void main(String[] args)      {            // Declaring integer arrays and          // initializing them with custom values          int arr1[] = { 1, 2, 3 };          int arr2[] = { 1, 2, 3 };            // Comparing arrays if equal or not          // using == operator          if (arr1 == arr2)                // Print statement              System.out.println("Same");          else                // Print statement              System.out.println("Not same");      }  } |

**Output**

Not same

**Output explanation:**In Java, [arrays are first class objects](https://www.geeksforgeeks.org/g-fact-65/). In the above program, arr1 and arr2 are two references to two different objects. So when we compare arr1 and arr2, two reference variables are compared, therefore we get the output as “Not Same”.

### How to Compare Array Contents?

A simple way is to run a loop and compare elements one by one. Java provides a direct method *Arrays.equals()* to compare two arrays. Actually, there is a list of equals() methods in the Arrays class for different primitive types (int, char, ..etc) and one for Object type (which is the base of all classes in Java).

**Example:**

* Java

|  |
| --- |
| // Java Program to Check If Two Arrays Are Equal  // Using equals() method of Arrays class    // Importing required classes  import java.util.Arrays;    // Main class  class GFG {        // Main driver method      public static void main(String[] args)      {            // Declaring integer arrays          int arr1[] = { 1, 2, 3 };          int arr2[] = { 1, 2, 3 };            // Checking if above two arrays are equal          // using equals() method          if (Arrays.equals(arr1, arr2))                // Print statement if arrays are equal              System.out.println("Same");          else                // Print statement if arrays are equal              System.out.println("Not same");      }  } |

**Output**

Same

**Output explanation:**As seen above, the Arrays.equals() works fine and compares arrays contents. Now the question, what if the arrays contain arrays inside them or some other references which refer to different objects but have the same values. For example, refer to the below program as follows.

### ****How to Deep Compare Array Contents?****

**Example 1-A:**

* Java

|  |
| --- |
| // Java Program to Check If Two Arrays Are Equal  // Using equals() method of Arrays class    // Importing required classes  import java.util.Arrays;    // Main class  class GFG {        // Main driver method      public static void main(String[] args)      {          // Declaring and initializing integer arrays          // having same values            // Array 1          int inarr1[] = { 1, 2, 3 };          // // Array 2          int inarr2[] = { 1, 2, 3 };            // // Array 1 contains only one element          Object[] arr1 = { inarr1 };          // Array 2 also contains only one element          Object[] arr2 = { inarr2 };            // Checking if arrays are equal or not          // using equals() method          if (Arrays.equals(arr1, arr2))                // Print statement if arrays are same              System.out.println("Same");          else                // Print statement if arrays are not same              System.out.println("Not same");      }  } |

**Output**

Not same

**Output explanation:**So [*Arrays.equals()*](https://www.geeksforgeeks.org/java-util-arrays-equals-java-examples/) is not able to do a deep comparison. Java provides another method for this [Arrays.deepEquals()](https://www.geeksforgeeks.org/java-util-arrays-deepequals-java/) which does the deep comparison.

**Example 1-B:**

* Java

|  |
| --- |
| // Java Program to Check If Two Arrays Are Equal  // Using deepEquals() method of Arrays class    // Importing required classes  import java.util.Arrays;    // Main class  class GFG {        // Main driver method      public static void main(String[] args)      {            // Declaring and initializing integer arrays          // having same values            // Array 1          int inarr1[] = { 1, 2, 3 };          // Array 2          int inarr2[] = { 1, 2, 3 };            // Array 1 contains only one element          Object[] arr1 = { inarr1 };          // Array 2 also contains only one element          Object[] arr2 = { inarr2 };            // Checking if arrays are equal or not          // using deepEquals() method          if (Arrays.deepEquals(arr1, arr2))                // Print statement if arrays are same              System.out.println("Same");          else                // Print statement if arrays are not same              System.out.println("Not same");      }  } |

**Output**

Same

What is enum

## **Enums**

An enum is a special "class" that represents a group of **constants** (unchangeable variables, like final variables).

To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a comma. Note that they should be in uppercase letters:

### Example

enum Level {

LOW,

MEDIUM,

HIGH

}

You can access enum constants with the **dot** syntax:

Level myVar = Level.MEDIUM;

# Variable Arguments (Varargs) in Java

* **Difficulty Level :** [Easy](https://www.geeksforgeeks.org/easy/)
* **Last Updated :** 08 Apr, 2022

Variable Arguments (Varargs) in Java is a method that takes a variable number of arguments. Variable Arguments in Java simplifies the creation of methods that need to take a variable number of arguments.

### Need of Java Varargs

* Until JDK 4, we cant declare a method with variable no. of arguments. If there is any change in the number of arguments, we have to declare a new method. This approach increases the length of the code and reduces readability.
* Before JDK 5, variable-length arguments could be handled in two ways. One uses an overloaded method(one for each), and another puts the arguments into an array and then passes this array to the method. Both of them are potentially error-prone and require more code.
* To resolve these problems, Variable Arguments (Varargs) were introduced in JDK 5. From JDK 5 onwards, we can declare a method with a variable number of arguments. Such types of methods are called Varargs methods. The varargs feature offers a simpler, better option.

**Syntax of Varargs**

Internally, the Varargs method is implemented by using the single dimensions arrays concept. Hence, in the Varargs method, we can differentiate arguments by using Index. A variable-length argument is specified by three periods or dots(…).

***For Example,***

public static void fun(int ... a)

{

// method body

}

This syntax tells the compiler that fun( ) can be called with zero or more arguments. As a result, here, a is implicitly declared as an array of type int[].

Below is a code snippet for illustrating the above concept :

* Java

|  |
| --- |
| // Java program to demonstrate varargs    class Test1 {      // A method that takes variable      // number of integer arguments.      static void fun(int... a)      {          System.out.println("Number of arguments: "                             + a.length);            // using for each loop to display contents of a          for (int i : a)              System.out.print(i + " ");          System.out.println();      }        // Driver code      public static void main(String args[])      {          // Calling the varargs method with          // different number of parameters            // one parameter          fun(100);              // four parameters          fun(1, 2, 3, 4);              // no parameter            fun();      }  } |

**Output**

Number of arguments: 1

100

Number of arguments: 4

1 2 3 4

Number of arguments: 0

What is tokenizing?

String tokenization is a process where a string is broken into several parts. Each part is called a token. For example, if “I am going” is a string, the discrete parts—such as “I”, “am”, and “going”—are the tokens. [Java](https://www.developer.com/java/) provides ready classes and methods to implement the tokenization process. They are quite handy to convey a specific semantics or contextual meaning to several individual parts of a string. This is particularly useful for text processing where you need to break a string into several parts and use each part as an element for individual processing. In a nutshell, tokenization is useful in any situation where you need to disorganize a string into individual parts; something to achieve with the part for the whole and whole for the part concept. This article provides information for a comprehensive understanding of the background concepts and its implementation in Java.

### [Learn JAVA and Start your Free Trial today!](https://o1.qnsr.com/cgi/r?;n=203;c=1662809;s=9481;x=7936;f=201903221139440;u=j;z=TIMESTAMP;k=https://www.pluralsight.com/browse/software-development/java?clickid=Q812P21nJxyJRsb0UfQwQyYMUkl2ftRLEVXK1w0&irgwc=1&mpid=1382528&utm_source=impactradius&utm_medium=digital_affiliate&utm_campaign=1382528&aid=7010a000001xAKZAA2)

## **String Tokenization with StringTokenizer**

A token or an individual element of a string can be filtered during infusion, meaning we can define the semantics of a token when extracting discrete elements from a string. For example, in a string say, “Hi! I am good. How about you?”, sometimes we may need to treat each word as a token or, at other times a set of words collectively as a token. So, a token basically is a flexible term and does not necessarily meant to be an atomic part, although it may be atomic according to the discretion of the context. For example, the keywords of a language are atomic according to the lexical analysis of the language, but they may typically be non-atomic and convey different meaning under a different context.

|  |
| --- |
| **Note:** Parsing is an important part of language processing. It is a process to resolve a language statement into several parts (tokens) and describe their roles. Tokenization is the process that breaks a statement into tokens, but parsing has a wider connotation; parsing can bring out the essence of the tokens through syntax and semantic roles defined by the grammar of the language. A complete description is beyond the scope of this article. Parsing has several uses; one of them is in designing a compiler. |

org.mano.example;

**import** java.util.StringTokenizer;

**public class** Main {

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

StringTokenizer st1 = **new** StringTokenizer("Hi!

I am good. How about you?");

**for** (**int** i = 1; st1.hasMoreTokens(); i++)

System.**out**.println("Token "+i+":

"+st1.nextToken());

}

}

The tokens are:

Token 1: Hi!

Token 2: I

Token 3: am

Token 4: good.

Token 5: How

Token 6: about

Token 7: you?

What is serialization

**Serialization** is the process of converting a data object—a combination of code and data represented within a region of data storage—into a series of bytes that saves the state of the object in an easily transmittable form. In this serialized form, the data can be delivered to another data store (such as an [**in-memory computing platform**](https://hazelcast.com/products/in-memory-computing-platform/)), application, or some other destination.

### What do you do if only parts of the object have to be serialized?

We mark all the properties of the object which should not be serialized as transient. Transient attributes in an object are not serialized. Area in the previous example is a calculated value. It is unnecessary to serialize and deserialize. We can calculate it when needed. In this situation, we can make the variable transient. Transient variables are not serialized. (transient int area;)

//Modified Rectangle class

class Rectangle implements Serializable {

public Rectangle(int length, int breadth) {

this.length = length;

this.breadth = breadth;

area = length \* breadth;

}

int length;

int breadth;

transient int area;

}

If you run the program again, you would get following output

System.out.println(rectangle.length);// 5

System.out.println(rectangle.breadth);// 6

System.out.println(rectangle.area);// 0

A hierarchy of objects is no impediment to using Java Serialization, as the latter can cope with arbitrary object graphs - and yes, serializing an object using Java Serialization will serialize all objects it refers to (unless that reference is marked transient). Assuming that's what you want, serializing the hierarchy is as simple as:

try (ObjectOutputStream oos = new ObjectOutputStream(new BufferedOutputStream(new FileOutputStream(filename)))) {

oos.write(rootScreen);

}

and reading as simple as:

try (ObjectInputStream ois = new ObjectInputStream(new BufferedInputStream(new FileInputStream(filename)))) {

return (GameScreen) ois.readObject();

}

Why do we need collections in Java?

Java Collection Framework **enables the user to perform various data manipulation operations like storing data, searching, sorting, insertion, deletion, and updating of data on the group of elements**.20-Jun-2022

**Add():**Add objects to collection

**Iterator():**Return an iterator over collection

**Clear():**Removes all elements from the collecti...

**RetainAll():**Retains elements in the collection

What are the important methods that are declared in the collection interface?

### What are the important methods that are declared in the Collection Interface?

Most important methods declared in the collection interface are the methods to add and remove an element. add method allows adding an element to a collection and delete method allows deleting an element from a collection.

size() methods returns number of elements in the collection. Other important methods defined as part of collection interface are shown below.

interface Collection<E> extends Iterable<E>

{

boolean add(E paramE);

boolean remove(Object paramObject);

int size();

boolean isEmpty();

void clear();

boolean contains(Object paramObject);

boolean containsAll(Collection<?> paramCollection);

boolean addAll(Collection<? extends E> paramCollection);

boolean removeAll(Collection<?> paramCollection);

boolean retainAll(Collection<?> paramCollection);

Iterator<E> iterator();

//A NUMBER OF OTHER METHODS AS WELL..

}

How do you iterate around an ArrayList using iterator?

# Java Program to Iterate over an ArrayList

In this example, we will learn to iterate over the elements of an arraylist in Java.

To understand this example, you should have the knowledge of the following [Java programming](https://www.programiz.com/java-programming) topics:

* [Java ArrayList](https://www.programiz.com/java-programming/arraylist)
* [Java for Loop](https://www.programiz.com/java-programming/for-loop)
* [Java for-each Loop](https://www.programiz.com/java-programming/enhanced-for-loop)
* [Java ListIterator Interface](https://www.programiz.com/java-programming/listiterator)

## Example 1: Iterate through ArrayList using for loop

import java.util.ArrayList;

class Main {

public static void main(String[] args) {

// Creating an array list

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

languages.add("Java");

languages.add("JavaScript");

languages.add("Python");

System.out.println("ArrayList: " + languages);

// Using for loop

System.out.println("Iterating over ArrayList using for loop: ");

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

System.out.print(languages.get(i));

System.out.print(", ");

}

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

ArrayList: [Java, JavaScript, Python]

Iterating over ArrayList using for loop:

Java, JavaScript, Python,

In the above example, we have created an arraylist named languages. Here, we have used the for loop to access each element of the arraylist.

## Example 2: Iterate through ArrayList using for-each loop

import java.util.ArrayList;

class Main {

public static void main(String[] args) {

// Creating an array list

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

languages.add("Java");

languages.add("JavaScript");

languages.add("Python");

System.out.println("ArrayList: " + languages);

// Using forEach loop

System.out.println("Iterating over ArrayList using for-each loop:");

for(String language : languages) {

System.out.print(language);

System.out.print(", ");

}

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

ArrayList: [Java, JavaScript, Python]

Iterating over ArrayList using for-each loop:

Java, JavaScript, Python,

Here, we have used the for-each loop to iterate over the ArrayList and print each element.

## Example 3: Iterate over ArrayList using listIterator()

import java.util.ArrayList;

import java.util.ListIterator;

class Main {

public static void main(String[] args) {

// Creating an ArrayList

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

numbers.add(1);

numbers.add(3);

numbers.add(2);

System.out.println("ArrayList: " + numbers);

// Creating an instance of ListIterator

ListIterator<Integer> iterate = numbers.listIterator();

System.out.println("Iterating over ArrayList:");

while(iterate.hasNext()) {

System.out.print(iterate.next() + ", ");

}

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

ArrayList: [1, 3, 2]

Iterating over ArrayList:

1, 3, 2,

How do you sort an ArrayList?

## **ArrayList**

In [Java](https://www.javatpoint.com/java-tutorial), **ArrayList** is a class of Collections framework that is defined in the java.util package. It inherits the AbstractList class. It dynamically stores the elements. The advantage of ArrayList is that it has no size limit. It is more flexible than the traditional array. It may have duplicate elements. We can also use all the methods of List interface because it implements the **List** interface.

We can sort an ArrayList in two ways ascending and descending order. The Collections class provides two methods to sort an [ArrayList in Java](https://www.javatpoint.com/arraylist).

54.3M

1.2K

Features of Java - Javatpoint

* **sort()**
* **reverseOrder()**

### Collections.sort() Method

An ArrayList can be sorted by using the **sort()** method of the [Collections class in Java](https://www.javatpoint.com/java-collections-class). It accepts an object of ArrayList as a parameter to be sort and returns an ArrayList sorted in the ascending order according to the natural ordering of its elements.

**Syntax**

1. **public** **static** <T **extends** Comparable<? **super** T>> **void** sort(List<T> list)

**Remember:** All elements in the ArrayList must be mutually comparable, else it throws **ClassCastException.** Here, mutually comparable means the list must have the same type of elements. For example, consider the snippet of the code:

1. //creating an instance of ArrayList that contains String type elements
2. ArrayList<String> list = **new** ArrayList<String>();
3. list.add("Computer");
4. list.add(123);
5. list.add("Hard Disk");
6. list.add("DRAM");

In the above example, we see that a list has four elements out of which three elements are of String type and one is Integer type. The three elements that are in String are mutually comparable but the element that is of Integer type is not comparable with the other three. Hence, the list must have the same type of elements.

### Collections.reverseOrder() Method

If we want to sort ArrayList in **descending** order, [Java Collections](https://www.javatpoint.com/collections-in-java) class provides **reverseOrder()** method. It allows us to sort the ArrayList in **reverse-lexicographic** order.

**Syntax**

1. **public** **static** <T> Comparator<T> reverseOrder()

It returns a comparator that imposes the reverse of the natural ordering on a collection of objects that implement the Comparable interface.

Remember that we do not directly invoke the reverseOrder() method. We use it along with the [**Collection.sort()**](https://www.javatpoint.com/java-collections-sort-method) method, as follows.

1. Collections.sort(objectOfArrayList, Collections.reverseOrder());

Therefore, the sorting of ArrayList in descending order done in two steps, first the ArrayList sorts the data in ascending order, after that the sorted data is reversed by the **reverseOrder()** method.

Let's create programs that sort ArrayList in ascending order.

### Sort ArrayList in Ascending Order

In the following example, we have created an ArrayList of type String and added some elements into it. After that we have invoked sort() method of the Collections class and passed the object of the ArrayList class i.e., list that sorts the elements in the ascending order.

**SortArrayListExample1.java**

1. **import** java.util.\*;
2. **public** **class** SortArrayListExample1
3. {
4. **public** **static** **void** main(String args[])
5. {
6. // creating object of ArrayList class
7. ArrayList<String> list = **new** ArrayList<String>();
8. // adding elements to the ArrayList
9. list.add("Volkswagen");
10. list.add("Toyota");
11. list.add("Porsche");
12. list.add("Ferrari");
13. list.add("Mercedes-Benz");
14. list.add("Audi");
15. list.add("Rolls-Royce");
16. list.add("BMW");
17. // printing the unsorted ArrayList
18. System.out.println("Before Sorting: "+ list);
19. // Sorting ArrayList in ascending Order
20. Collections.sort(list);
21. // printing the sorted ArrayList
22. System.out.println("After Sorting: "+ list);
23. }
24. }

**Output:**

Before Sorting: [Volkswagen, Toyota, Porsche, Ferrari, Mercedes-Benz, Audi, Rolls-Royce, BMW]

After Sorting: [Audi, BMW, Ferrari, Mercedes-Benz, Porsche, Rolls-Royce, Toyota, Volkswagen]

Let's see another example that sorts an ArrayList of Integer type.

How do you sort elements in an ArrayList using comparable interface?

. We can simply implement Comparator without affecting the original User-defined class. To sort an ArrayList using Comparator we need to override the compare() method provided by comparator interface. After rewriting the compare() method we need to call collections.sort() method like below.

**Syntax:**

Collections.sort(list, comparator)

**Parameters:**

* **list:**List which should be sorted based on the comparator.
* **comparator:**Comparator class instance

**Returns:**It sorts the list and does not return anything.

**Example**

* Java

|  |  |
| --- | --- |
| // Java program to Sort ArrayList using Comparator    import java.util.\*;    // create the Shop class  class Shop {      int ProductNo;      String name;      int stock;        // constructor      Shop(int ProductNo, String name, int stock)      {          this.ProductNo = ProductNo;          this.name = name;          this.stock = stock;      }  }    // creates the comparator for comparing stock value  class StockComparator implements Comparator<Shop> {        // override the compare() method      public int compare(Shop s1, Shop s2)      {          if (s1.stock == s2.stock)              return 0;          else if (s1.stock > s2.stock)              return 1;          else              return -1;      }  }    class GFG {      public static void main(String[] args)      {          // create the ArrayList object          ArrayList<Shop> s = new ArrayList<Shop>();          s.add(new Shop(218, "Pen", 520));          s.add(new Shop(223, "Pencil", 213));          s.add(new Shop(423, "Books", 101));          s.add(new Shop(512, "Toy", 59));          s.add(new Shop(723, "Bottle", 10));            System.out.println("before sorting");          for (Shop shop : s) {              System.out.println(shop.stock + " " + shop.name                                 + " " + shop.ProductNo);          }          System.out.println();  How do you sort elements in an ArrayList using comparator interface?   We can simply implement Comparator without affecting the original User-defined class. To sort an ArrayList using Comparator we need to override the compare() method provided by comparator interface. After rewriting the compare() method we need to call collections.sort() method like below.  **Syntax:**  Collections.sort(list, comparator)  **Parameters:**   * **list:**List which should be sorted based on the comparator. * **comparator:**Comparator class instance   **Returns:**It sorts the list and does not return anything.  **Example**   * Java  |  | | --- | | // Java program to Sort ArrayList using Comparator    import java.util.\*;    // create the Shop class  class Shop {      int ProductNo;      String name;      int stock;        // constructor      Shop(int ProductNo, String name, int stock)      {          this.ProductNo = ProductNo;          this.name = name;          this.stock = stock;      }  }    // creates the comparator for comparing stock value  class StockComparator implements Comparator<Shop> {        // override the compare() method      public int compare(Shop s1, Shop s2)      {          if (s1.stock == s2.stock)              return 0;          else if (s1.stock > s2.stock)              return 1;          else              return -1;      }  }    class GFG {      public static void main(String[] args)      {          // create the ArrayList object          ArrayList<Shop> s = new ArrayList<Shop>();          s.add(new Shop(218, "Pen", 520));          s.add(new Shop(223, "Pencil", 213));          s.add(new Shop(423, "Books", 101));          s.add(new Shop(512, "Toy", 59));          s.add(new Shop(723, "Bottle", 10));            System.out.println("before sorting");          for (Shop shop : s) {              System.out.println(shop.stock + " " + shop.name                                 + " " + shop.ProductNo);          }          System.out.println();            System.out.println(              "After sorting(sorted by Stock)");            // call the sort function          Collections.sort(s, new StockComparator());          for (Shop shop : s) {              System.out.println(shop.stock + " " + shop.name                                 + " " + shop.ProductNo);          }      }  } |   **Output**  before sorting  520 Pen 218  213 Pencil 223  101 Books 423  59 Toy 512  10 Bottle 723  After sorting(sorted by Stock)  10 Bottle 723  59 Toy 512  101 Books 423  213 Pencil 223  520 Pen 218            System.out.println(              "After sorting(sorted by Stock)");            // call the sort function          Collections.sort(s, new StockComparator());          for (Shop shop : s) {              System.out.println(shop.stock + " " + shop.name                                 + " " + shop.ProductNo);          }      }  } |

**Output**

before sorting

520 Pen 218

213 Pencil 223

101 Books 423

59 Toy 512

10 Bottle 723

After sorting(sorted by Stock)

10 Bottle 723

59 Toy 512

101 Books 423

213 Pencil 223

520 Pen 218

What is the difference between Set and sortedSet interfaces?

The [SortedSet](https://www.geeksforgeeks.org/sortedset-java-examples/) is a sub-interface, which is available in [java.util.package](https://www.geeksforgeeks.org/java-util-package-java/) which extends the Set interface. This interface contains the methods inherited from the Set interface. For example, headSet, tailSet, subSet, Comparator, first, last, and many more.

**Example**

* Java

|  |
| --- |
| // Java program to Illustrate SortedSet    // Importing utility classes  import java.util.\*;    // Main class  class GFG {        // Main driver method      public static void main(String[] args)      {            // Creating an instance of SortedSet          // String type          SortedSet<String> ts = new TreeSet<String>();            // Adding elements into the TreeSet          // using add()          ts.add("Sravan");          ts.add("Ojaswi");          ts.add("Bobby");          ts.add("Rohith");          ts.add("Gnanesh");          ts.add("Devi2");            // Adding the duplicate element          // again simply using add() method          ts.add("Sravan");            // Print and display TreeSet          System.out.println(ts);            // Removing items from TreeSet          // using remove() method          ts.remove("Ojaswi");            // Display message          System.out.println("Iterating over set:");            // Iterating over TreeSet items          Iterator<String> i = ts.iterator();            // Condition holds true till there is single element          // remaining in the object          while (i.hasNext())                // Printing elements              System.out.println(i.next());      }  } |

**Output**

[Bobby, Devi2, Gnanesh, Ojaswi, Rohith, Sravan]

Iterating over set:

Bobby

Devi2

Gnanesh

Rohith

Sravan

Can you give examples of classes that implement the Set interface?

classes that implement Set

In order to use functionalities of the Set interface, we can use these classes: **HashSet**. **LinkedHashSet**. **EnumSet**.

# NavigableSet in Java with Examples

* **Difficulty Level :** [Easy](https://www.geeksforgeeks.org/easy/)
* **Last Updated :** 28 Jun, 2021

NavigableSet represents a navigable set in [Java Collection Framework](https://www.geeksforgeeks.org/collections-in-java-2/). The NavigableSet interface inherits from the [SortedSet interface](https://www.geeksforgeeks.org/sortedset-java-examples/). It behaves like a SortedSet with the exception that we have navigation methods available in addition to the sorting mechanisms of the SortedSet.   
For example, the NavigableSet interface can navigate the set in reverse order compared to the order defined in SortedSet. A NavigableSet may be accessed and traversed in either ascending or descending order. The classes that implement this interface are, [TreeSet](https://www.geeksforgeeks.org/treeset-in-java/) and [ConcurrentSkipListSet](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListSet.html)



Here, E is the type of elements maintained by this set.

**All Superinterfaces:**

Collection<E>, Iterable<E>, [Set<E>](https://www.geeksforgeeks.org/set-in-java/), [SortedSet<E>](https://www.geeksforgeeks.org/sortedset-java-examples/)

**All Known Implementing Classes:**

[ConcurrentSkipListSet](https://www.geeksforgeeks.org/concurrentskiplistset-in-java-with-examples/), [TreeSet<E>](https://www.geeksforgeeks.org/treeset-in-java-with-examples/)

**Declaration:**The NavigableSet is declared as

*public interface NavigableSet<E>  extends SortedSet<E>*

**Creating NavigableSet Objects**

Since NavigableSet is an interface, objects cannot be created of the type NavigableSet. We always need a class that extends this list in order to create an object. And also, after the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the NavigableSet. This type-safe set can be defined as:

*// Obj is the type of the object to be stored in NavigableSet*

*NavigableSet<Obj> set = new TreeSet<Obj> ();*

**Example:**

* Java

|  |
| --- |
| // Java program to demonstrate  // the working of NavigableSet  import java.util.NavigableSet;  import java.util.TreeSet;    public class NavigableSetDemo  {      public static void main(String[] args)      {          NavigableSet<Integer> ns = new TreeSet<>();          ns.add(0);          ns.add(1);          ns.add(2);          ns.add(3);          ns.add(4);          ns.add(5);          ns.add(6);            // Get a reverse view of the navigable set          NavigableSet<Integer> reverseNs = ns.descendingSet();            // Print the normal and reverse views          System.out.println("Normal order: " + ns);          System.out.println("Reverse order: " + reverseNs);            NavigableSet<Integer> threeOrMore = ns.tailSet(3, true);          System.out.println("3 or  more:  " + threeOrMore);          System.out.println("lower(3): " + ns.lower(3));          System.out.println("floor(3): " + ns.floor(3));          System.out.println("higher(3): " + ns.higher(3));          System.out.println("ceiling(3): " + ns.ceiling(3));            System.out.println("pollFirst(): " + ns.pollFirst());          System.out.println("Navigable Set:  " + ns);            System.out.println("pollLast(): " + ns.pollLast());          System.out.println("Navigable Set:  " + ns);            System.out.println("pollFirst(): " + ns.pollFirst());          System.out.println("Navigable Set:  " + ns);            System.out.println("pollFirst(): " + ns.pollFirst());          System.out.println("Navigable Set:  " + ns);            System.out.println("pollFirst(): " + ns.pollFirst());          System.out.println("Navigable Set:  " + ns);            System.out.println("pollFirst(): " + ns.pollFirst());          System.out.println("pollLast(): " + ns.pollLast());      }  } |

**Output**

Normal order: [0, 1, 2, 3, 4, 5, 6]

Reverse order: [6, 5, 4, 3, 2, 1, 0]

3 or more: [3, 4, 5, 6]

lower(3): 2

floor(3): 3

higher(3): 4

ceiling(3): 3

pollFirst(): 0

Navigable Set: [1, 2, 3, 4, 5, 6]

pollLast(): 6

Navigable Set: [1, 2, 3, 4, 5]

pollFirst(): 1

Navigable Set: [2, 3, 4, 5]

pollFirst(): 2

Navigable Set: [3, 4, 5]

pollFirst(): 3

Navigable Set: [4, 5]

pollFirst(): 4

pollLast(): 5

. What is difference between Map and sortedMap?

## **Explain the concepts of Map and SortedMap interface.**

Keys will be mapped to their values using Map object. Map allows no duplicate values. The keys in a map objects must be unique. Java collection framework allows implementing Map interface in three classes namely, HashMap, TreeMap and LinkedHashMap.  
  
SortedMap is a special interface for maintaining all the elements in a sorted order. This interface extends Map interface. It maintains all the elements in ascending order. The sorting process is performed on the map keys. It has two additional methods than Map interface. They are firstKey() and lastKey(). Method firstKey() returns the first value available currently in the map, where as the lastKey() returns the last value available currently in the map.

## **Collections class declaration**

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

1. **public** **class** Collections **extends** Object

|  |  |  |  |
| --- | --- | --- | --- |
| **SN** | **Modifier & Type** | **Methods** | **Descriptions** |
| 1) | static <T> boolean | [addAll()](https://www.javatpoint.com/java-collections-addall-method) | It is used to adds all of the specified elements to the specified collection. |
| 2) | static <T> Queue<T> | [asLifoQueue()](https://www.javatpoint.com/java-collections-aslifoqueue-method) | It returns a view of a Deque as a Last-in-first-out (LIFO) Queue. |
| 3) | static <T> int | [binarySearch()](https://www.javatpoint.com/java-collections-binarysearch-method) | It searches the list for the specified object and returns their position in a sorted list. |
| 4) | static <E> Collection<E> | [checkedCollection()](https://www.javatpoint.com/java-collections-checkedcollection-method) | It is used to returns a dynamically typesafe view of the specified collection. |
| 5) | static <E> List<E> | [checkedList()](https://www.javatpoint.com/java-collections-checkedlist-method) | It is used to returns a dynamically typesafe view of the specified list. |
| 6) | static <K,V> Map<K,V> | [checkedMap()](https://www.javatpoint.com/java-collections-checkedmap-method) | It is used to returns a dynamically typesafe view of the specified map. |
| 7) | static <K,V> NavigableMap<K,V> | [checkedNavigableMap()](https://www.javatpoint.com/java-collections-checkednavigablemap-method) | It is used to returns a dynamically typesafe view of the specified navigable map. |
| 8) | static <E> NavigableSet<E> | [checkedNavigableSet()](https://www.javatpoint.com/java-collections-checkednavigableset-method) | It is used to returns a dynamically typesafe view of the specified navigable set. |
| 9) | static <E> Queue<E> | [checkedQueue()](https://www.javatpoint.com/java-collections-checkedqueue-method) | It is used to returns a dynamically typesafe view of the specified queue. |
| 10) | static <E> Set<E> | [checkedSet()](https://www.javatpoint.com/java-collections-checkedset-method) | It is used to returns a dynamically typesafe view of the specified set. |
| 11) | static <K,V> SortedMap<K,V> | [checkedSortedMap()](https://www.javatpoint.com/java-collections-checkedsortedmap-method) | It is used to returns a dynamically typesafe view of the specified sorted map. |
| 12) | static <E> SortedSet<E> | [checkedSortedSet()](https://www.javatpoint.com/java-collections-checkedsortedset-method) | It is used to returns a dynamically typesafe view of the specified sorted set. |
| 13) | static <T> void | [copy()](https://www.javatpoint.com/java-collections-copy-method) | It is used to copy all the elements from one list into another list. |
| 14) | static boolean | [disjoint()](https://www.javatpoint.com/java-collections-disjoint-method) | It returns true if the two specified collections have no elements in common. |
| 15) | static <T> Enumeration<T> | [emptyEnumeration()](https://www.javatpoint.com/java-collections-emptyenumeration-method) | It is used to get an enumeration that has no elements. |
| 16) | static <T> Iterator<T> | [emptyIterator()](https://www.javatpoint.com/java-collections-emptyiterator-method) | It is used to get an Iterator that has no elements. |
| 17) | static <T> List<T> | [emptyList()](https://www.javatpoint.com/java-collections-emptylist-method) | It is used to get a List that has no elements. |
| 18) | static <T> ListIterator<T> | [emptyListIterator()](https://www.javatpoint.com/java-collections-emptylistiterator-method) | It is used to get a List Iterator that has no elements. |
| 19) | static <K,V> Map<K,V> | [emptyMap()](https://www.javatpoint.com/java-collections-emptymap-method) | It returns an empty map which is immutable. |
| 20) | static <K,V> NavigableMap<K,V> | [emptyNavigableMap()](https://www.javatpoint.com/java-collections-emptynavigablemap-method) | It returns an empty navigable map which is immutable. |
| 21) | static <E> NavigableSet<E> | [emptyNavigableSet()](https://www.javatpoint.com/java-collections-emptynavigableset-method) | It is used to get an empty navigable set which is immutable in nature. |
| 22) | static <T> Set<T> | [emptySet()](https://www.javatpoint.com/java-collections-emptyset-method) | It is used to get the set that has no elements. |
| 23) | static <K,V> SortedMap<K,V> | [emptySortedMap()](https://www.javatpoint.com/java-collections-emptysortedmap-method) | It returns an empty sorted map which is immutable. |
| 24) | static <E> SortedSet<E> | [emptySortedSet()](https://www.javatpoint.com/java-collections-emptysortedset-method) | It is used to get the sorted set that has no elements. |
| 25) | static <T> Enumeration<T> | [enumeration()](https://www.javatpoint.com/java-collections-enumeration-method) | It is used to get the enumeration over the specified collection. |
| 26) | static <T> void | [fill()](https://www.javatpoint.com/java-collections-fill-method) | It is used to replace all of the elements of the specified list with the specified elements. |
| 27) | static int | [frequency()](https://www.javatpoint.com/java-collections-frequency-method) | It is used to get the number of elements in the specified collection equal to the specified object. |
| 28) | static int | [indexOfSubList()](https://www.javatpoint.com/java-collections-indexofsublist-method) | It is used to get the starting position of the first occurrence of the specified target list within the specified source list. It returns -1 if there is no such occurrence in the specified list. |
| 29) | static int | [lastIndexOfSubList()](https://www.javatpoint.com/java-collections-lastindexofsublist-method) | It is used to get the starting position of the last occurrence of the specified target list within the specified source list. It returns -1 if there is no such occurrence in the specified list. |
| 30) | static <T> ArrayList<T> | [list()](https://www.javatpoint.com/java-collections-list-method) | It is used to get an array list containing the elements returned by the specified enumeration in the order in which they are returned by the enumeration. |
| 31) | static <T extends Object & Comparable<? super T>> T | [max()](https://www.javatpoint.com/java-collections-max-method) | It is used to get the maximum value of the given collection, according to the natural ordering of its elements. |
| 32) | static <T extends Object & Comparable<? super T>> T | [min()](https://www.javatpoint.com/java-collections-min-method) | It is used to get the minimum value of the given collection, according to the natural ordering of its elements. |
| 33) | static <T> List<T> | [nCopies()](https://www.javatpoint.com/java-collections-ncopies-method) | It is used to get an immutable list consisting of **n** copies of the specified object. |
| 34) | static <E> Set<E> | [newSetFromMap()](https://www.javatpoint.com/java-collections-newsetfrommap-method) | It is used to return a set backed by the specified map. |
| 35) | static <T> boolean | [replaceAll()](https://www.javatpoint.com/java-collections-replaceall-method) | It is used to replace all occurrences of one specified value in a list with the other specified value. |
| 36) | static void | [reverse()](https://www.javatpoint.com/java-collections-reverse-method) | It is used to reverse the order of the elements in the specified list. |
| 37) | static <T> Comparator<T> | [reverseOrder()](https://www.javatpoint.com/java-collections-reverseorder-method) | It is used to get the comparator that imposes the reverse of the natural ordering on a collection of objects which implement the Comparable interface. |
| 38) | static void | [rotate()](https://www.javatpoint.com/java-collections-rotate-method) | It is used to rotate the elements in the specified list by a given distance. |
| 39) | static void | [shuffle()](https://www.javatpoint.com/java-collections-shuffle-method) | It is used to randomly reorders the specified list elements using a default randomness. |
| 40) | static <T> Set<T> | [singleton()](https://www.javatpoint.com/java-collections-singleton-method) | It is used to get an immutable set which contains only the specified object. |
| 41) | static <T> List<T> | [singletonList()](https://www.javatpoint.com/java-collections-singletonlist-method) | It is used to get an immutable list which contains only the specified object. |
| 42) | static <K,V> Map<K,V> | [singletonMap()](https://www.javatpoint.com/java-collections-singletonmap-method) | It is used to get an immutable map, mapping only the specified key to the specified value. |
| 43) | static <T extends Comparable<? super T>>void | [sort()](https://www.javatpoint.com/java-collections-sort-method) | It is used to sort the elements presents in the specified list of collection in ascending order. |
| 44) | static void | [swap()](https://www.javatpoint.com/java-collections-swap-method) | It is used to swap the elements at the specified positions in the specified list. |
| 45) | static <T> Collection<T> | [synchronizedCollection()](https://www.javatpoint.com/java-collections-synchronizedcollection-method) | It is used to get a synchronized (thread-safe) collection backed by the specified collection. |
| 46) | static <T> List<T> | [synchronizedList()](https://www.javatpoint.com/java-collections-synchronizedlist-method) | It is used to get a synchronized (thread-safe) collection backed by the specified list. |
| 47) | static <K,V> Map<K,V> | [synchronizedMap()](https://www.javatpoint.com/java-collections-synchronizedmap-method) | It is used to get a synchronized (thread-safe) map backed by the specified map. |
| 48) | static <K,V> NavigableMap<K,V> | [synchronizedNavigableMap()](https://www.javatpoint.com/java-collections-synchronizednavigablemap-method) | It is used to get a synchronized (thread-safe) navigable map backed by the specified navigable map. |
| 49) | static <T> NavigableSet<T> | [synchronizedNavigableSet()](https://www.javatpoint.com/java-collections-synchronizednavigableset-method) | It is used to get a synchronized (thread-safe) navigable set backed by the specified navigable set. |
| 50) | static <T> Set<T> | [synchronizedSet()](https://www.javatpoint.com/java-collections-synchronizedset-method) | It is used to get a synchronized (thread-safe) set backed by the specified set. |
| 51) | static <K,V> SortedMap<K,V> | [synchronizedSortedMap()](https://www.javatpoint.com/java-collections-synchronizedsortedmap-method) | It is used to get a synchronized (thread-safe) sorted map backed by the specified sorted map. |
| 52) | static <T> SortedSet<T> | [synchronizedSortedSet()](https://www.javatpoint.com/java-collections-synchronizedsortedset-method) | It is used to get a synchronized (thread-safe) sorted set backed by the specified sorted set. |
| 53) | static <T> Collection<T> | [unmodifiableCollection()](https://www.javatpoint.com/java-collections-unmodifiablecollection-method) | It is used to get an unmodifiable view of the specified collection. |
| 54) | static <T> List<T> | [unmodifiableList()](https://www.javatpoint.com/java-collections-unmodifiablelist-method) | It is used to get an unmodifiable view of the specified list. |
| 55) | static <K,V> Map<K,V> | [unmodifiableMap()](https://www.javatpoint.com/java-collections-unmodifiablemap-method) | It is used to get an unmodifiable view of the specified map. |
| 56) | static <K,V> NavigableMap<K,V> | [unmodifiableNavigableMap()](https://www.javatpoint.com/java-collections-unmodifiablenavigablemap-method) | It is used to get an unmodifiable view of the specified navigable map. |
| 57) | static <T> NavigableSet<T> | [unmodifiableNavigableSet()](https://www.javatpoint.com/java-collections-unmodifiablenavigableset-method) | It is used to get an unmodifiable view of the specified navigable set. |
| 58) | static <T> Set<T> | [unmodifiableSet()](https://www.javatpoint.com/java-collections-unmodifiableset-method) | It is used to get an unmodifiable view of the specified set. |
| 59) | static <K,V> SortedMap<K,V> | [unmodifiableSortedMap()](https://www.javatpoint.com/java-collections-unmodifiablesortedmap-method) | It is used to get an unmodifiable view of the specified sorted map. |
| 60 | static <T> SortedSet<T> | [unmodifiableSortedSet()](https://www.javatpoint.com/java-collections-unmodifiablesortedset-method) | It is used to get an unmodifiable view of the specified sorted set. |

## **Java Collections Example**

1. **import** java.util.\*;
2. **public** **class** CollectionsExample {
3. **public** **static** **void** main(String a[]){
4. List<String> list = **new** ArrayList<String>();
5. list.add("C");
6. list.add("Core Java");
7. list.add("Advance Java");
8. System.out.println("Initial collection value:"+list);
9. Collections.addAll(list, "Servlet","JSP");
10. System.out.println("After adding elements collection value:"+list);
11. String[] strArr = {"C#", ".Net"};
12. Collections.addAll(list, strArr);
13. System.out.println("After adding array collection value:"+list);
14. }
15. }

Output:

Initial collection value:[C, Core Java, Advance Java]

After adding elements collection value:[C, Core Java, Advance Java, Servlet, JSP]

After adding array collection value:[C, Core Java, Advance Java, Servlet, JSP, C#, .Net]

## **Java Collections Example: max()**

1. **import** java.util.\*;
2. **public** **class** CollectionsExample {
3. **public** **static** **void** main(String a[]){
4. List<Integer> list = **new** ArrayList<Integer>();
5. list.add(46);
6. list.add(67);
7. list.add(24);
8. list.add(16);
9. list.add(8);
10. list.add(12);
11. System.out.println("Value of maximum element from the collection: "+Collections.max(list));
12. }
13. }

Output:

Value of maximum element from the collection: 67

What is the difference between synchronized and concurrent collections in Java?

## Synchronized Collections vs Concurrent Collections in Java

The synchronized collections classes, [Hashtable](http://java67.blogspot.com/2012/08/5-difference-between-hashtable-hashmap-Java-collection.html) and [Vector](http://java67.blogspot.com/2016/02/how-to-convert-vector-to-array-in-java.html), and the synchronized wrapper classes, Collections.synchronizedMap() and Collections.synchronizedList(), provides a basic conditionally thread-safe implementation of Map and List.  
  
However, several factors make them unsuitable for use in highly concurrent applications, most importantly their single collection-wide lock is an impediment to scalability and it often becomes necessary to lock a collection for a considerable time during iteration to prevent [ConcurrentModificationException](http://java67.blogspot.com/2015/10/how-to-solve-concurrentmodificationexception-in-java-arraylist.html).

Explain about the new concurrent collections in Java?

concurrent package includes a number of additions to the Java Collections Framework. These are most easily categorized by the collection interfaces provided: **BlockingQueue defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue**.

Explain about copyonwrite concurrent collections approach?

**CopyOnWriteArrayList class** is introduced in JDK 1.5, which implements the [**List interface**](https://www.geeksforgeeks.org/list-interface-java-examples/). It is an enhanced version of [**ArrayList**](https://www.geeksforgeeks.org/arraylist-in-java/)in which all modifications (add, set, remove, etc) are implemented by making a fresh copy. It is found in **java.util.concurrent** package. It is a data structure created to be used in a concurrent environment.



**Here are few points about CopyOnWriteArrayList:**

* As the name indicates, CopyOnWriteArrayList creates a Cloned copy of underlying ArrayList, for every update operation at a certain point both will be synchronized automatically, which is taken care of by JVM. Therefore, there is no effect for threads that are performing read operation.
* It is costly to use because for every update operation a cloned copy will be created. Hence, CopyOnWriteArrayList is the best choice if our frequent operation is read operation.
* The underlined data structure is a grow-able array.
* It is a thread-safe version of ArrayList.
* Insertion is preserved, duplicates, null, and heterogeneous Objects are allowed.
* The main important point about CopyOnWriteArrayList is the [Iterator](https://www.geeksforgeeks.org/iterators-in-java/) of CopyOnWriteArrayList can not perform remove operation otherwise we get Run-time exception saying **UnsupportedOperationException.**add() and set() methods on CopyOnWriteArrayList iterator also throws **UnsupportedOperationException.**Also Iterator of CopyOnWriteArrayList will never throw **ConcurrentModificationException**.

**Declaration:**

*public class CopyOnWriteArrayList<E> extends Object implements List<E>, RandomAccess, Cloneable, Serializable*

Here, E is the type of elements held in this collection.

**Note:** The class implements **Serializable**, **Cloneable**, **Iterable<E>**, **Collection<E>**, [List<E>](https://www.geeksforgeeks.org/list-interface-java-examples/), **RandomAccess** interfaces.

**Constructors:**

**1. CopyOnWriteArrayList()**: Creates an empty list.

CopyOnWriteArrayList c = new CopyOnWriteArrayList();

**2. CopyOnWriteArrayList(Collection obj)**: Creates a list containing the elements of the specified collection, in the order, they are returned by the collection’s iterator.

CopyOnWriteArrayList c = new CopyOnWriteArrayList(Collection obj);

**3. CopyOnWriteArrayList(Object[] obj);**: Creates a list holding a copy of the given array.

CopyOnWriteArrayList c = new CopyOnWriteArrayList(Object[] obj);

**Example:**

* Java

|  |
| --- |
| // Java program to illustrate  // CopyOnWriteArrayList class  import java.util.\*;  import java.util.concurrent.CopyOnWriteArrayList;    public class ConcurrentDemo extends Thread {        static CopyOnWriteArrayList<String> l          = new CopyOnWriteArrayList<String>();        public void run()      {          // Child thread trying to          // add new element in the          // Collection object          l.add("D");      }        public static void main(String[] args)          throws InterruptedException      {          l.add("A");          l.add("B");          l.add("c");            // We create a child thread          // that is going to modify          // ArrayList l.          ConcurrentDemo t = new ConcurrentDemo();          t.run();            Thread.sleep(1000);            // Now we iterate through          // the ArrayList and get          // exception.          Iterator itr = l.iterator();          while (itr.hasNext()) {              String s = (String)itr.next();              System.out.println(s);              Thread.sleep(1000);          }          System.out.println(l);      }  } |

**Output**

A

B

c

D

[A, B, c, D]

What is compareandswap approach?

**Compare and swap** is a technique used when designing concurrent algorithms.  The approach is to compare the actual value of the variable to the expected value of the variable and if the actual value matches the expected value, then swap the actual value of the variable for the new value passed in.

For understanding the algorithm, one must have the basic knowledge of [Concurrency](https://www.geeksforgeeks.org/java-concurrency-yield-sleep-and-join-methods/) and[Multithread](https://www.geeksforgeeks.org/multithreading-in-java/) in Java.

**Working of the Algorithm:**

It is like we know that this variable should be 1, and we want to change it to 2. Since this is a **multithreaded environment,** we know that others might be working on the same variable. So we should first check if the value of the variable is 1 as we thought and if yes, then we change it to 2. If we see that the variable is 3 now, then that means someone else is working on it and so let us not touch it at this time.

**Check then Act approach**:-

The most common condition for concurrency problems is the**“check then act”**approach. The “check then act”  occurs when the code first checks the value of a variable and then acts based on that value. Here is a simple example:

public boolean lock() {

if(!locked) {

lock = true;

return true;

}

return false;

}

*Above, pseudocode Implies that, if it is already locked, you don’t need to lock again. So you first****check****and only if required,****act****.*

The above code is not safe as If two or more threads might have access to lock() function and do the check at the same time , then above**lock()** function would not be guaranteed to work because all thread would lock the resource and use it as it’s own.

**Let’s elaborate it :**

There is **Thread A**and **Thread B**, thread B may check locked at any time between thread A has checked locked and seen it to be false then, both thread A and thread B may see locked as being false, and both will then act based on that information.

The above problem can be resolved by making the entire block of code as synchronized. Then only one thread (thread A or thread B) going to the check and act at one time, Only after the thread that got into the code completes its check, and it’s an act then another thread gets to try. Now there will be no misunderstanding between threads.

**Example for synchronized code :**

class GFG {

private boolean locked = false;

public synchronized boolean lock() {

if(!locked) {

locked = true;

return true;

}

return false;

}

}

Now the lock() method is synchronized so only one thread can execute it at a time on the same lock() function.

**Atomic Operation**

After Java 5, we don’t have to implement or write a synchronized block with the check and act code anymore, Java 5 offers this support via **java.util.concurrent.atomic:**a toolkit of classes used for lock-free, thread-safe programming on single variables.

**AtomicBoolean** makes sure that only one thread can read it at a time.

Here is an example showing how to implement the lock() method using**AtomicBoolean** :

public static class MyLock {

private AtomicBoolean locked = new AtomicBoolean(false);

public boolean lock() {

return locked.compareAndSet(false, true);

}

}

Notice how the locked variable is no longer a boolean but an AtomicBoolean. This class has a **compareAndSet()**function which will compare the value of the AtomicBoolean instance to an expected value, and if has the expected value, it swaps the value with a new value. In this case it compares the value of locked to false and if it is false it sets the new value of the AtomicBoolean to true.

The compareAndSet() method returns true if the value was swapped, and false if not.

So there is many other Atomic Variable like:

* [**AtomicInteger**](https://www.geeksforgeeks.org/atomicinteger-compareandset-method-in-java-with-examples/)**:**This variable lets you update an int value atomically.
* [**AtomicLong:**](https://www.geeksforgeeks.org/atomiclong-compareandset-method-in-java-with-examples/)Long with thread-safe “Compare and Swap” functionality.
* [**AtomicReference:**](https://www.geeksforgeeks.org/atomicreference-compareandset-method-in-java-with-examples/)This variable provides an object reference variable which can be read and written atomically.
* **AtomicIntegerArray, AtomicLongArray, and AtomicReferenceArray**

**Example of Atomic Integer:**

* Java

|  |
| --- |
| // Java Program to demonstrates  // the compareAndSet() function    import java.util.concurrent.atomic.AtomicInteger;    public class GFG {      public static void main(String args[])      {            // Initially value as 0          AtomicInteger val = new AtomicInteger(0);            // Prints the updated value          System.out.println("Previous value: "                             + val);            // Checks if previous value was 0          // and then updates it          boolean res = val.compareAndSet(0, 6);            // Checks if the value was updated.          if (res)              System.out.println("The value was"                                 + " updated and it is "                                 + val);          else              System.out.println("The value was "                                 + "not updated");      }  } |

**Output**

Previous value: 0

The value was updated and it is 6

What is a lock? How is it different from using synchronized approach?

Major difference between lock and synchronized: **with locks, you can release and acquire the locks in any order.** **with synchronized, you can release the locks only in the order it was acquired**.

# **he default capacity of Collection Elements: ArrayList, Vector, HashSet, Hashtable, and HashMap.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Collection | Java 8 | | Before Java 8 | |
| Initial capacity | Load factor | Initial capacity | Load factor |
| ArrayList | 0 (lazily initialized to 10) |  | 10 |  |
| Vector | 10 |  | 10 |  |
| HashSet | 16 | 0.75 | 16 | 0.75 |
| HashMap | 16 | 0.75 | 16 | 0.75 |
| Hashtable | 16 | 0.75 | 11 | 0.75 |

The load factor is **the measure that decides when to increase the capacity of the Map**. The default load factor is 75% of the capacity. The threshold of a HashMap is approximately the product of current capacity and load factor. Rehashing is the process of re-calculating the hash code of already stored entries.

When does a Java collection throw UnsupportedOperationException?

## UnsupportedOperationException Example

Here’s an example of an UnsupportedOperationException thrown when an object is attempted to be added to an unmodifiable List:

import java.util.Arrays;

import java.util.List;

public class UnsupportedOperationExceptionExample {

public static void main(String[] args) {

String array[] = {"a", "b", "c"};

List<String> list = Arrays.asList(array);

list.add("d");

}

}

Since the Arrays.asList() method returns a fixed-size list, attempting to modify it either by adding or removing elements throws an UnsupportedOperationException.

Running the above code throws the exception:

Exception in thread "main" java.lang.UnsupportedOperationException

at java.base/java.util.AbstractList.add(AbstractList.java:153)

at java.base/java.util.AbstractList.add(AbstractList.java:111)

at UnsupportedOperationExceptionExample.main(UnsupportedOperationExceptionExample.java:8)

What are Generics?

**Generics** means **parameterized types**. The idea is to allow type (Integer, String, … etc., and user-defined types) to be a parameter to methods, classes, and interfaces.

### What are Generics?

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

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

Consider the class below:

class MyList {

private List<String> values;

void add(String value) {

values.add(value);

}

void remove(String value) {

values.remove(value);

}

}

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

MyList myList = new MyList();

myList.add("Value 1");

myList.add("Value 2");

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

##### **Example with Generics**

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

class MyListGeneric<T> {

private List<T> values;

void add(T value) {

values.add(value);

}

void remove(T value) {

values.remove(value);

}

T get(int index) {

return values.get(index);

}

}

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

myListString.add("Value 1");

myListString.add("Value 2");

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

myListInteger.add(1);

myListInteger.add(2);

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

* Cannot Instantiate Generic Types with Primitive Types.
* Cannot Create Instances of Type Parameters.
* Cannot Declare Static Fields Whose Types are Type Parameters.
* Cannot Use Casts or instanceof With Parameterized Types.
* Cannot Create Arrays of Parameterized Types.

Whenever you want to restrict the type parameter to subtypes of a particular class you can use the bounded type parameter. If you just specify a type (class) as bounded parameter, only sub types of that particular class are accepted by the current generic class.

You can declare a bound parameter just by extending the required class with the type-parameter, within the angular braces as −

class Sample <T extends Number>

## **Example**

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

In the following Java example the generic class Sample restricts the type parameter to the sub classes of the Number classes using the bounded parameter.

class Sample <T extends Number>{

   T data;

   Sample(T data){

      this.data = data;

   }

   public void display() {

      System.out.println("Data value is: "+this.data);

   }

}

public class BoundsExample {

   public static void main(String args[]) {

      Sample<Integer> obj1 = new Sample<Integer>(20);

      obj1.display();

      Sample<Double> obj2 = new Sample<Double>(20.22d);

      obj2.display();

      Sample<Float> obj3 = new Sample<Float>(125.332f);

      obj3.display();

   }

}

## **Output**

Data value is: 20

Data value is: 20.22

Data value is: 125.332

What is the need for threads in Java?

Threads **allows a program to operate more efficiently by doing multiple things at the same time**. Threads can be used to perform complicated tasks in the background without interrupting the main program.

What are the different states of a thread?

**A thread can be in one of the following states:**

* NEW. A thread that has not yet started is in this state.
* RUNNABLE. A thread executing in the Java virtual machine is in this state.
* BLOCKED. A thread that is blocked waiting for a monitor lock is in this state.
* WAITING. ...
* TIMED\_WAITING. ...
* TERMINATED.

What is priority of a thread? How do you change the priority of a thread?

# **Java Thread setPriority() method**

The **setPriority()** method of thread class is used to change the thread's priority. Every thread has a priority which is represented by the integer number between 1 to 10.

Thread class provides 3 constant properties:

1. **public static int MIN\_PRIORITY:** It is the maximum priority of a thread. The value of it is 1.
2. **public static int NORM\_PRIORITY:** It is the normal priority of a thread. The value of it is 5.
3. **public static int MAX\_PRIORITY:** It is the minimum priority of a thread. The value of it is 10.

We can also set the priority of thread between 1 to 10. This priority is known as custom priority or user defined priority.

## **Syntax**

1. **public** **final** **void** setPriority(**int** a)

## **Parameter**

**a**: It is the priority to set this thread to.

7.5M

76

Competitive questions on Structures

## **Return**

It does not return any value.

## **Exception**

**IllegalArgumentException:** This exception throws if the priority is not in the range MIN\_PRIORITY to MAX\_PRIORITY.

**SecurityException:** This exception throws if the current thread cannot modify this thread.

## **Example 1: Maximum Priority Thread**

1. **public** **class** JavaSetPriorityExp1 **extends** Thread
2. {
3. **public** **void** run()
4. {
5. System.out.println("Priority of thread is: "+Thread.currentThread().getPriority());
6. }
7. **public** **static** **void** main(String args[])
8. {
9. // creating one thread
10. JavaSetPriorityExp1 t1=**new** JavaSetPriorityExp1();
11. // print the maximum priority of this thread
12. t1.setPriority(Thread.MAX\_PRIORITY);
13. // call the run() method
14. t1.start();
15. }
16. }

**[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=JavaSetPriorityExp1" \t "_blank)**

**Output:**

Priority of thread is: 10

What is executorservice?

[*ExecutorService*](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ExecutorService.html) is a JDK API that simplifies running tasks in asynchronous mode. Generally speaking, ExecutorService automatically provides a pool of threads and an API for assigning tasks to it.

How do you check whether an executionservice task executed successfully?

There isn't a clean way to check if all Runnables are done if you use ExecutorService.execute(Runnable). Unless you build a mechanism to do so in the Runnable itself (which is sloppy in my opinion).

**Instead:**  
Use ExecutorService.submit(Runnable). This method will return a Future<?> which is a handle to the result of a Runnable. Using Futures provides a clean way to check results.

All you have to do is maintain a list of Futures that you submit, and then you can iterate over the whole list of Futures and either:  
  A) wait for all the futures to be done in a blocking way or  
  B) check if all the futures are done in a non-blocking way.

Here is a code example:

List<Future<?>> futures = new ArrayList<Future<?>>();

ExecutorService exec = Executors.newFixedThreadPool(5);

// Instead of using exec.execute() use exec.submit()

// because it returns a monitorable future

while((item = stack.pollFirst()) != null){

Runnable worker = new Solider(this, item);

Future<?> f = exec.submit(worker);

futures.add(f);

}

// A) Await all runnables to be done (blocking)

for(Future<?> future : futures)

future.get(); // get will block until the future is done

// B) Check if all runnables are done (non-blocking)

boolean allDone = true;

for(Future<?> future : futures){

allDone &= future.isDone(); // check if future is done

}

What is callable? How do you execute a callable from executionservice?

**1. Important interfaces**

**1.1. Callable**

In [Java concurrency](https://howtodoinjava.com/java-concurrency-tutorial/), **Callable** represents a task that returns a result. [**Executor**](https://howtodoinjava.com/java/multi-threading/executor-service-example/) can run callable tasks – concurrently.

Since [Java 8](https://howtodoinjava.com/java-8-tutorial/), it is a [functional interface](https://howtodoinjava.com/java/stream/functional-interface-tutorial/) and can therefore be used as the assignment target for a [lambda expression](https://howtodoinjava.com/java8/lambda-expressions/) or [method reference](https://howtodoinjava.com/java8/lambda-method-references-example/).

|  |
| --- |
| Callable.java |
| @FunctionalInterface  public interface Callable<V> {  /\*\*   \* Computes a result, or throws an exception if unable to do so.   \*   \* @return computed result   \* @throws Exception if unable to compute a result   \*/    V call() throws Exception;  } |

Synchronization in java is **the capability to control the access of multiple threads to any shared resource**. In the Multithreading concept, multiple threads try to access the shared resources at a time to produce inconsistent results. The synchronization is necessary for reliable communication between threads

**TestSynchronizedBlock1.java**

1. **class** Table
2. {
3. **void** printTable(**int** n){
4. **synchronized**(**this**){//synchronized block
5. **for**(**int** i=1;i<=5;i++){
6. System.out.println(n\*i);
7. **try**{
8. Thread.sleep(400);
9. }**catch**(Exception e){System.out.println(e);}
10. }
11. }
12. }//end of the method
13. }
15. **class** MyThread1 **extends** Thread{
16. Table t;
17. MyThread1(Table t){
18. **this**.t=t;
19. }
20. **public** **void** run(){
21. t.printTable(5);
22. }
24. }
25. **class** MyThread2 **extends** Thread{
26. Table t;
27. MyThread2(Table t){
28. **this**.t=t;
29. }
30. **public** **void** run(){
31. t.printTable(100);
32. }
33. }
35. **public** **class** TestSynchronizedBlock1{
36. **public** **static** **void** main(String args[]){
37. Table obj = **new** Table();//only one object
38. MyThread1 t1=**new** MyThread1(obj);
39. MyThread2 t2=**new** MyThread2(obj);
40. t1.start();
41. t2.start();
42. }
43. }

**Output:**

5

10

15

20

25

100

200

300

400

500

**static methods can be synchronized**. But you have one lock per class. when the java class is loaded coresponding java.

**Joining Threads in Java**

* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
* Last Updated : 17 Feb, 2021

**java.lang.Thread** class provides the join() method which allows one thread to wait until another thread completes its execution. If **t** is a Thread object whose thread is currently executing, then **t.join()** will make sure that **t** is terminated before the next instruction is executed by the program.  
If there are multiple threads calling the join() methods that means overloading on join allows the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.  
There are three overloaded join functions.

1. **join():**It will put the current thread on wait until the thread on which it is called is dead. If thread is interrupted then it will throw InterruptedException.  
   **Syntax:**
2. public final void join()
3. **join(long millis)**:It will put the current thread on wait until the thread on which it is called is dead or wait for specified time (milliseconds).  
   **Syntax:**
4. public final synchronized void join(long millis)
5. **join(long millis, int nanos):**It will put the current thread on wait until the thread on which it is called is dead or wait for specified time (milliseconds + nanos).  
   **Syntax:**
6. public final synchronized void join(long millis, int nanos)

|  |
| --- |
| // Java program to explain the  // concept of joining a thread.  import java.io.\*;    // Creating thread by creating the  // objects of that class  class ThreadJoining extends Thread  {      @Override      public void run()      {          for (int i = 0; i < 2; i++)          {              try              {                  Thread.sleep(500);                  System.out.println("Current Thread: "                          + Thread.currentThread().getName());              }                catch(Exception ex)              {                  System.out.println("Exception has" +                                  " been caught" + ex);              }              System.out.println(i);          }      }  }    class GFG  {      public static void main (String[] args)      {            // creating two threads          ThreadJoining t1 = new ThreadJoining();          ThreadJoining t2 = new ThreadJoining();          ThreadJoining t3 = new ThreadJoining();            // thread t1 starts          t1.start();            // starts second thread after when          // first thread t1 has died.          try          {              System.out.println("Current Thread: "                    + Thread.currentThread().getName());              t1.join();          }            catch(Exception ex)          {              System.out.println("Exception has " +                                  "been caught" + ex);          }            // t2 starts          t2.start();            // starts t3 after when thread t2 has died.          try          {              System.out.println("Current Thread: "                   + Thread.currentThread().getName());              t2.join();          }            catch(Exception ex)          {              System.out.println("Exception has been" +                                      " caught" + ex);          }            t3.start();      }  } |

1. output:
2. Current Thread: main
3. Current Thread: Thread-0
4. 0
5. Current Thread: Thread-0
6. 1
7. Current Thread: main
8. Current Thread: Thread-1
9. 0
10. Current Thread: Thread-1
11. 1
12. Current Thread: Thread-2
13. 0
14. Current Thread: Thread-2
15. 1

 What is a deadlock?

Deadlock in Java is **a condition where two or more threads are blocked forever, waiting for each other**. This usually happens when multiple threads need the same locks but obtain them in different orders. Multithreaded Programming in Java suffers from the deadlock situation because of the synchronized keyword.1

What are the important methods in Java for inter-thread communication?

**Inter-thread Communication in Java**

* wait()
* notify()
* notifyAll()

Description

The **java.lang.Object.wait()** causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object. In other words, this method behaves exactly as if it simply performs the call wait(0).

The current thread must own this object's monitor. The thread releases ownership of this monitor and waits until another thread notifies threads waiting on this object's monitor to wake up either through a call to the notify method or the notifyAll method. The thread then waits until it can re-obtain ownership of the monitor and resumes execution.

This method should only be called by a thread that is the owner of this object's monitor. See the notify method for a description of the ways in which a thread can become the owner of a monitor.

Declaration

Following is the declaration for **java.lang.Object.wait()** method

public final void wait()

Parameters

NA

Return Value

This method does not return a value.

Exception

* **IllegalMonitorStateException** − if the current thread is not the owner of the object's monitor.
* **InterruptedException** − if another thread has interrupted the current thread. The interrupted status of the current thread is cleared when this exception is thrown.

Example

The following example shows the usage of lang.Object.wait() method.

[Live Demo](http://tpcg.io/1J3AXr)

package com.tutorialspoint;

import java.util.Collections;

import java.util.LinkedList;

import java.util.List;

public class ObjectDemo extends Object {

private List synchedList;

public ObjectDemo() {

// create a new synchronized list to be used

synchedList = Collections.synchronizedList(new LinkedList());

}

// method used to remove an element from the list

public String removeElement() throws InterruptedException {

synchronized (synchedList) {

// while the list is empty, wait

while (synchedList.isEmpty()) {

System.out.println("List is empty...");

synchedList.wait();

System.out.println("Waiting...");

}

String element = (String) synchedList.remove(0);

return element;

}

}

// method to add an element in the list

public void addElement(String element) {

System.out.println("Opening...");

synchronized (synchedList) {

// add an element and notify all that an element exists

synchedList.add(element);

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

synchedList.notifyAll();

System.out.println("notifyAll called!");

}

System.out.println("Closing...");

}

public static void main(String[] args) {

final ObjectDemo demo = new ObjectDemo();

Runnable runA = new Runnable() {

public void run() {

try {

String item = demo.removeElement();

System.out.println("" + item);

} catch (InterruptedException ix) {

System.out.println("Interrupted Exception!");

} catch (Exception x) {

System.out.println("Exception thrown.");

}

}

};

Runnable runB = new Runnable() {

// run adds an element in the list and starts the loop

public void run() {

demo.addElement("Hello!");

}

};

try {

Thread threadA1 = new Thread(runA, "A");

threadA1.start();

Thread.sleep(500);

Thread threadA2 = new Thread(runA, "B");

threadA2.start();

Thread.sleep(500);

Thread threadB = new Thread(runB, "C");

threadB.start();

Thread.sleep(1000);

threadA1.interrupt();

threadA2.interrupt();

} catch (InterruptedException x) {

}

}

}

Let us compile and run the above program, this will produce the following result −

List is empty...

List is empty...

Opening...

New Element:'Hello!'

notifyAll called!

Closing...

Waiting...

Hello!

Waiting...

List is empty...

Interrupted Exception!

Functional programming is a paradigm that allows programming using expressions i.e. declaring functions, passing functions as arguments and using functions as statements (rightly called expressions in Java8).

**Example #2: Lambda Expressions**

* Lambda expression used to represent a method interface with an expression.
* It helps to iterate, filtering and extracting data from collections.
* Lambda expression interface implementation is a [functional interface](https://www.educba.com/functional-interface-in-java/).
* It reduces a lot of code.
* Lambda expression treated as a function so java compiler can’t create .class

**Syntax:**

(arguments) ->

{

//code for implementation

}

Arguments: argument-list can be have values or no values

Example: arguments1, arguments2, arguments3,……

->: Joins code implementation and arguments.

**a. Lambda expression with a single argument**

**Syntax:**

(argument1) ->

{

//code for implementation

}

**Example – AreaOfSquare.java**

**Code:**

package com.lambda;//creating a package

interface Square{ //creating interface for quare

public int getArea(int side); //create a method for get area

}

public class AreaOfSquare{ //As we are working with Lambda expression so no need to implement interface of square

public static void main(String[] args) {

Square area=(side)->{ // Lambda expression with only one argument.

Return side\*side; //returning area

};

System.out.println(“Area of Square=>”+area.getArea(10)); //printing area by calling interface getArea method

}

}

**Output:**

A stream is **a sequence of objects that supports various methods which can be pipelined to produce the desired result**. The features of Java stream are – A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.

ava provides I/O Streams to read and write data where, a Stream represents an input source or an output destination which could be a file, i/o devise, other program etc.

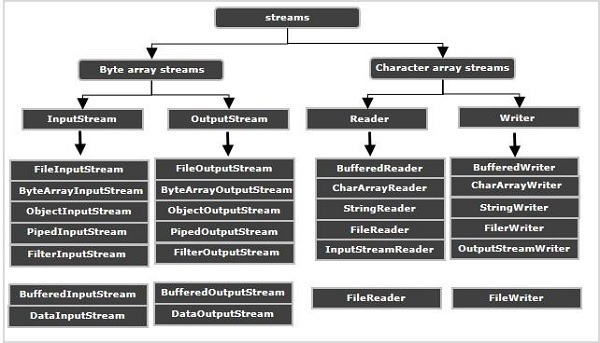
In general, a Stream will be an input stream or, an output stream.

* **InputStream** − This is used to read data from a source.
* **OutputStream** − This is used to write data to a destination.

Based on the data they handle there are two types of streams −

* **Byte Streams** − These handle data in bytes (8 bits) i.e., the byte stream classes read/write data of 8 bits. Using these you can store characters, videos, audios, images etc.
* **Character Streams** − These handle data in 16 bit Unicode. Using these you can read and write text data only.

Following diagram illustrates all the input and output Streams (classes) in Java.



Standard Streams

In addition to above mentioned classes Java provides 3 standard streams representing the input and, output devices.

* **Standard Input** − This is used to read data from user through input devices. keyboard is used as standard input stream and represented as System.in.
* **Standard Output** − This is used to project data (results) to the user through output devices. A computer screen is used for standard output stream and represented as System.out.
* **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as System.err.

Example

Following Java program reads the data from user using BufferedInputStream and writes it into a file using BufferedOutputStream.

import java.io.BufferedInputStream;

import java.io.BufferedOutputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class BufferedInputStreamExample {

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

      //Creating an BufferedInputStream object

      BufferedInputStream inputStream = new BufferedInputStream(System.in);

      byte bytes[] = new byte[1024];

      System.out.println("Enter your data ");

      //Reading data from key-board

      inputStream.read(bytes);

      //Creating BufferedOutputStream object

      FileOutputStream out= new FileOutputStream("D:/myFile.txt");

      BufferedOutputStream outputStream = new BufferedOutputStream(out);

      //Writing data to the file

      outputStream.write(bytes);

      outputStream.flush();

      System.out.println("Data successfully written in the specified file");

   }

}

Output

Enter your data

Hi welcome to Tutorialspoint ....

Data successfully written in the specified file

# Stream In Java

* **Difficulty Level :** [Medium](https://www.geeksforgeeks.org/medium/)
* **Last Updated :** 09 Oct, 2019

Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.  
The features of Java stream are –

* A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Different Operations On Streams-  
**Intermediate Operations:**

1. **map:**The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.  
   List number = Arrays.asList(2,3,4,5);  
   List square = number.stream().map(x->x\*x).collect(Collectors.toList());
2. **filter:** The filter method is used to select elements as per the Predicate passed as argument.  
   List names = Arrays.asList("Reflection","Collection","Stream");  
   List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
3. **sorted:** The sorted method is used to sort the stream.  
   List names = Arrays.asList("Reflection","Collection","Stream");  
   List result = names.stream().sorted().collect(Collectors.toList());

**Terminal Operations:**

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.  
   List number = Arrays.asList(2,3,4,5,3);  
   Set square = number.stream().map(x->x\*x).collect(Collectors.toSet());
2. **forEach:** The forEach method is used to iterate through every element of the stream.  
   List number = Arrays.asList(2,3,4,5);  
   number.stream().map(x->x\*x).forEach(y->System.out.println(y));
3. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.  
   The reduce method takes a BinaryOperator as a parameter.

List number = Arrays.asList(2,3,4,5);  
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

Here ans variable is assigned 0 as the initial value and i is added to it .

**Program to demonstrate the use of Stream**

|  |
| --- |
| //a simple program to demonstrate the use of stream in java  import java.util.\*;  import java.util.stream.\*;    class Demo  {    public static void main(String args[])    {        // create a list of integers      List<Integer> number = Arrays.asList(2,3,4,5);        // demonstration of map method      List<Integer> square = number.stream().map(x -> x\*x).                             collect(Collectors.toList());      System.out.println(square);        // create a list of String      List<String> names =                  Arrays.asList("Reflection","Collection","Stream");        // demonstration of filter method      List<String> result = names.stream().filter(s->s.startsWith("S")).                            collect(Collectors.toList());      System.out.println(result);        // demonstration of sorted method      List<String> show =              names.stream().sorted().collect(Collectors.toList());      System.out.println(show);        // create a list of integers      List<Integer> numbers = Arrays.asList(2,3,4,5,2);        // collect method returns a set      Set<Integer> squareSet =           numbers.stream().map(x->x\*x).collect(Collectors.toSet());      System.out.println(squareSet);        // demonstration of forEach method      number.stream().map(x->x\*x).forEach(y->System.out.println(y));        // demonstration of reduce method      int even =         number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);        System.out.println(even);    }  } |

Output:

[4, 9, 16, 25]

[Stream]

[Collection, Reflection, Stream]

[16, 4, 9, 25]

4

9

16

25

6

**Important Points/Observations:**

1. A stream consists of source followed by zero or more intermediate methods combined together (pipelined) and a terminal method to process the objects obtained from the source as per the methods described.
2. Stream is used to compute elements as per the pipelined methods without altering the original value of the object.

This article is contributed by **Akash Ojha** .If you like GeeksforGeeks and would like to contribute, you can also write an article using [contribute.geeksforgeeks.org](http://www.contribute.geeksforgeeks.org/) or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Difference between intermediate and terminal operations in Java 8

[Java](https://www.tutorialspoint.com/questions/category/Java)[Server Side Programming](https://www.tutorialspoint.com/questions/category/Server-Side-Programming)[Programming](https://www.tutorialspoint.com/questions/category/Programming)

Stream is introduced in Java 8, it is only used for processing group of data not for the storting elements.. It does not modify the actual collection, they only provide the result as per the pipelined methods.

Stream api supports multiple operations and operations are divided into two parts −

* Intermediate Operation- These operations are used to pipeline other methods and to transform into the other streams. They don’t produce results because these operation does not invoke until the terminal operation gets executed. Below are the examples −
* sorted(Comparator<T>)
* peek(Consumer<T>)
* distinct()
* Terminal operations - These operations are used to produce results. They can’t be used for chaining the other methods. Below are the examples −
* forEach
* count
* toArray

| **Sr. No.** | **Key** | **Intermediate Operations** | **Terminal Operations** |
| --- | --- | --- | --- |
| 1 | Basic | These operations are used to pipeline other methods and to transform into the other streams | A terminal operation in Java is a method applied to a stream as the final step. |
| 2 | Return Type | They only return another stream. | They return final result. |
| 3 | Method | sorted(Comparator<T>) peek(Consumer<T>) distinct() | forEach count toArray |
| 4. | Use Case | These operations should be used to transform stream into another stream | They can be used to produce results. |

Example of Intermediate and Terminal Operation

public class Main {

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

      List<String> laptopList = new ArrayList();

      laptopList.add("DELL");

      laptopList.add("ACER");

      laptopList.add("HCL");

      // Intermediate operation

      laptopList.sort((p1, p2) -> p1.compareTo(p2));

      // Terminal Operation

      laptopList.forEach(a -> {

         System.out.println(a);

      });

   }

}

A terminal stream operation is **an operation that starts the internal iteration of the elements, calls all the listeners, and returns a result**. The call to the map() method of the Stream interface is a non-terminal operation. It merely sets a lambda expression on the stream which converts each element to lowercase.

## Terminal and Non-Terminal Operations

The Stream interface has a selection of terminal and non-terminal operations. A non-terminal stream operation is an operation that adds a listener to the stream without doing anything else. A terminal stream operation is an operation that starts the internal iteration of the elements, calls all the listeners, and returns a result.

Here is a Java Stream example which contains both a non-terminal and a terminal operation:

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Stream;

public class StreamExamples {

public static void main(String[] args) {

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

stringList.add("ONE");

stringList.add("TWO");

stringList.add("THREE");

Stream<String> stream = stringList.stream();

long count = stream

.map((value) -> { return value.toLowerCase(); })

.count();

System.out.println("count = " + count);

}

}

The call to the map() method of the Stream interface is a non-terminal operation. It merely sets a lambda expression on the stream which converts each element to lowercase. The map() method will be covered in more detail later on.

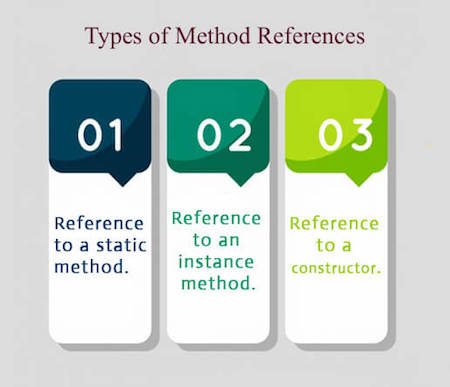
ava Method References

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference. In this tutorial, we are explaining method reference concept in detail.

Types of Method References

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.



1) Reference to a Static Method

You can refer to static method defined in the class. Following is the syntax and example which describe the process of referring static method in Java.

Syntax

1. ContainingClass::staticMethodName

Example 1

In the following example, we have defined a functional interface and referring a static method to it's functional method say().

1. **interface** Sayable{
2. **void** say();
3. }
4. **public** **class** MethodReference {
5. **public** **static** **void** saySomething(){
6. System.out.println("Hello, this is static method.");
7. }
8. **public** **static** **void** main(String[] args) {
9. // Referring static method
10. Sayable sayable = MethodReference::saySomething;
11. // Calling interface method
12. sayable.say();
13. }
14. }

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

Output:

Hello, this is static method.

Example 2

In the following example, we are using predefined functional interface Runnable to refer static method.

1. **public** **class** MethodReference2 {
2. **public** **static** **void** ThreadStatus(){
3. System.out.println("Thread is running...");
4. }
5. **public** **static** **void** main(String[] args) {
6. Thread t2=**new** Thread(MethodReference2::ThreadStatus);
7. t2.start();
8. }
9. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=MethodReference2)

Output:

Thread is running...

[Home](https://www.informit.com/) > [Articles](https://www.informit.com/articles/index.aspx) > [Programming](https://www.informit.com/articles/index.aspx?st=60206) > [Java](https://www.informit.com/articles/index.aspx?st=60209)

[**Java SE 8’s New Language Features, Part 1: Interface Default/Static Methods and Lambda Expressions**](https://www.informit.com/articles/article.aspx?p=2191423)

* Mar 25, 2014

[📄 Contents](https://www.informit.com/articles/article.aspx?p=2191423&seqNum=2)

[␡](https://www.informit.com/articles/article.aspx?p=2191423&seqNum=2)

1. [Interface Default and Static Methods](https://www.informit.com/articles/article.aspx?p=2191423)
2. Lambda Expressions and Functional Interfaces

* [⎙ Print](https://www.informit.com/articles/printerfriendly/2191423)
* [+ Share This](http://www.addthis.com/bookmark.php)

[< Back](https://www.informit.com/articles/article.aspx?p=2191423) **Page 2** of 2

**Like this article? We recommend**

[](https://www.informit.com/store/java-se8-for-the-really-impatient-a-short-course-on-9780321927767?w_ptgrevartcl=Java+SE+8%e2%80%99s+New+Language+Features,+Part+1:+Interface+Default/Static+Methods+and+Lambda+Expressions_2191423)[Java SE8 for the Really Impatient: A Short Course on the Basics](https://www.informit.com/store/java-se8-for-the-really-impatient-a-short-course-on-9780321927767?w_ptgrevartcl=Java+SE+8%e2%80%99s+New+Language+Features%2c+Part+1%3a+Interface+Default%2fStatic+Methods+and+Lambda+Expressions_2191423)  
  
[Learn More](https://www.informit.com/store/java-se8-for-the-really-impatient-a-short-course-on-9780321927767?w_ptgrevartcl=Java+SE+8%e2%80%99s+New+Language+Features%2c+Part+1%3a+Interface+Default%2fStatic+Methods+and+Lambda+Expressions_2191423) [Buy](https://www.informit.com/buy.aspx?isbn=9780321927767&w_ptgrevartcl=Java+SE+8%e2%80%99s+New+Language+Features%2c+Part+1%3a+Interface+Default%2fStatic+Methods+and+Lambda+Expressions_2191423)

**Lambda Expressions and Functional Interfaces**

A *lambda expression* (*lambda*) is a short-form replacement for an anonymous class. Lambdas simplify the use of interfaces that declare single abstract methods. Such interfaces are known as *functional interfaces*.

ℹ Note

Java 8 introduces a new FunctionalInterface annotation type that lets you annotate an interface as being functional. For example, the following code fragment shows how java.lang.Runnable is annotated as a functional interface:

@FunctionalInterface

public interface Runnable

{

public abstract void run();

}

A functional interface can define as many default and static methods as it requires. However, it must declare exactly one abstract method, or the compiler will complain that it isn't a functional interface.

Lambdas let you treat code as data. You can use lambdas to pass code to methods for subsequent execution. Listing 4 contrasts the traditional (cumbersome) approach to passing code via an anonymous class with the cleaner approach to passing code via a lambda.

**Listing 4 LambdaDemo.java (Version 1).**

import java.awt.EventQueue;

public class LambdaDemo

{

public static void main(String[] args)

{

Runnable r = new Runnable()

{

@Override

public void run()

{

System.out.println("Running");

}

};

EventQueue.invokeLater(r);

EventQueue.invokeLater(() -> System.out.println("Running"));

}

}