**https://javatechonline.com/java-interview-questions/**

**C is procedural oriented language (focus on tasks), C++ is a object oriented language (focus on class and objects)**

**C, C++ are platform dependent languages..**

**James Goslings- 1991**

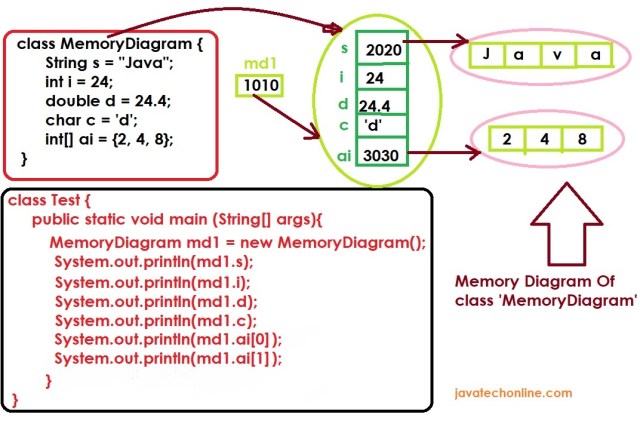
**What are the features of Java ?**

**Ans :** There are 12 important buzzwords(features) in java.

1. **Simple :** Easy to learn, most of the difficult features like pointers, multiple inheritance concepts have been removed from Java.
2. **Platform independent :** Write once run anywhere.
3. **Architecture Neutral :** If you perform any changes in your architecture of system, java program will still run without any issue.
4. **Portable :** We can migrate java code from windows to Linux easily.
5. **Secure :** Lets you creating applications that can’t be invaded from outside.
6. **Object Oriented :** From JDK 1.8 onwards Java has also incorporated functional(procedural) programming.
7. **Multi-threaded :** You can build applications with many concurrent threads of activity, resulting in highly responsive applications.

**Multi threading means handling multiple tasks simultaneously**

1. **Robust (Strong):** Having reliable programming habits for creating highly reliable applications. **–1. Early checking of errors**- 2. **java is strong language by using exceptions and memory management(garbage collections)**
2. **Distributed :** It facilitates users to create distributed applications in Java. To create distributed applications we use RMI, EJB, CORBA etc .
3. **Interpreted :** Java integrates the power of Compiled Languages with the flexibility of Interpreted Languages.
4. **High Performance :** Java code after compilation converts to bytecode which is highly optimized by the Java compiler, so that the Java virtual machine (JVM) can execute Java applications at full speed.
5. **Dynamic :** It supports dynamic loading of classes. It means class loading happens on demand.



# JVM Architecture & Class Loaders Java

### JVM Architecture & Class Loaders Java

**♦ Static Variables are stored in Method area**

**♦ Instance Variables are stored in Heap area**

**♦ Local Variables are stored in Stack area**

## JVM Memory Area

The division of total Memory area of JVM is in 5 parts :

### 1. Method Area :

In the method area, One area will be allocated for each JVM.  It will be created at JVM startup. Class level binary information & static variables reside in this area. Also Constant pools will be saved inside method area. Further It can be accessed by multiple threads simultaneously, therefore it is not thread-safe.

### 2. Heap Area :

One area will be allocated for each JVM. It will be created at JVM startup. Objects reside in this area. It can be accessed by multiple threads simultaneously, therefore it is also not thread-safe.

### 3. Stack Area :

It is available per thread unlike Method & Heap area as they are one per JVM. Each entry in stack is called Stack frame or activation record. Also it is thread-safe as it allocates one memory for each thread. Furthermore, Each stack frame has three parts : local variable array, operand stack and frame data.  
**Local Variable Array :** It contains values of local variables & method parameters.  
**Operand Stack :** JVM uses it as workspace, some instruction push the values to it & some pop from it & some other to performs arithmetic operations.  
**Frame Data :** It contains all symbolic references related to the method. It also contains reference of exception related to method.

### 4. PC Registers :

(Program Counter Registers) : Internally used by JVM. For every thread JVM creates a separate PC register. In brief PC register contains address of currently executing threads.

### 5. Native Method Stacks :

For every thread JVM creates a separate native method stack if its native method call.

**\*\*\*\*Note :** Method Area, Heap Area, Stack Area are also considered as important memory areas as programmers point of view. Method Area, Heap area are for per JVM whereas Stack area, PC register & the native method stack are for per thread.

**♦ Static Variables are stored in Method area**

**♦ Instance Variables are stored in Heap area**

**♦ Local Variables are stored in Stack area**

## Execution Engine

It is a central component of JVM and responsible for executing  .class files. It  mainly contains two parts : Interpreter & JIT compiler

### Interpreter :

Interpreter reads & interprets bytecode, converts it into machine code/native code line by line. Because of line by line performance of system goes down. Then JIT compiler comes into picture in jdk 1.1 version.

### JIT Compiler :

The primary purpose of JIT compiler is to improve performance. In fact, Internally it maintains a separate count for every method. Whenever JVM comes across any method call, first that method is interpreted normally by the interpreter and JIT compiler increments the corresponding count variable accordingly.  
This process continues for every method. Once if any method count reaches threshold value then JIT compiler identifies that the method is repeatedly used method. We also call that method as hotspot for the JIT compiler. Then JIT Compiler compiles that method immediately & generates corresponding native code. Next time JVM comes across that method call, then JVM uses native code directly & executes it instead of interpreting it once again so that performance of the system will be improved. However the threshold count varies from JVM to JVM.

### How is Java a platform independent language?

**Ans :**In general the compiler’s job in a programming language is to convert source code into machine understandable code( also called executable code). There is one more state of code between source code & machine understandable code in java which is called the byte-code. In fact converting java source code to machine understandable code is two step process. One is from source code to byte-code(.class file) which is done by compiler. Other is from byte-code to machine understandable code which is done by the JVM. However JVM is a software which comes automatically with JDK installation.

JVM is platform dependent. Therefore while converting from byte-code to executable code JVM makes it compatible with the platform to which it belongs. In this complete process there are two translators which makes this possible ie. compiler & JVM. Compiler can read source file(.java file). Similarly JVM can read byte-code(.class file). Because of the feature of platform in-dependency Java’s slogan in **Write Once Run Anywhere(WORA)**. Also below points are important to keep in mind.

**Java Source Code - platform independent**

**Java Bytecode - platform independent**

**Java Compiler - platform independent**

**Java Executable Code - platform dependent**

**Java Virtual Machine - platform dependent**

**Java Software(jdk) - platform dependent**

**Java Program - platform independent**

**Java Software Application - platform independent**

### What is JRE, JDK, JVM in java and what are the differences between them?

**Ans:**First of all let’s understand what are the usage of these terms in java. [JRE(Java Runtime Environment)](https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqd/index.html#:~:text=The%20JRE%20is%20the%20software,Implementation%20of%20the%20JVM) helps us to run the compiled code in java whereas [JDK(Java Development Kit)](https://en.wikipedia.org/wiki/Java_Development_Kit) helps us to compile & run the code. Hence if you want to run already compiled code such as a code in the form of jar/war/ear, only JRE is sufficient. On the other hand if want to compile & then run your code, you must have JDK in your system. JVM(Java Virtual machine) helps us to convert compiled code(byte code) into machine language code. JRE contains JVM and JDK contains JRE.

**JRE = Java API + JVM**

**JDK = Compiler + JRE**

**JVM = Interpreter + JIT**

JVM runs java program with the help of interpreter.  
JIT (Just in time compiler) helps interpreter to run fast when its slow.

**OBJECT:**

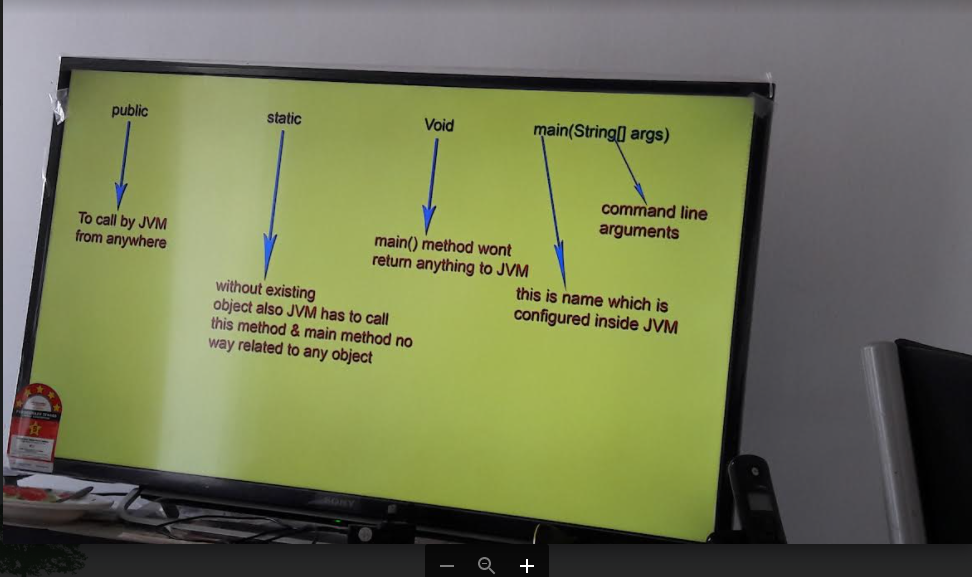
**Object is defined as a real world entity**

**It consists of state and behaviours**

**State is nothing but variables**

**Behaviours is nothing but Methods**

**Object is an instance of a class.**



**Class:**

**Class is blueprint/ prototype that defines the variables and methods of object.**

**Class in java determines how an object will behave and what the object will contain.**

**Ex: Animal, Human,Table**

**Class – dog**

**Variables – breed,size, age, color**

**Methods- eat(), sleep(), sit(),**

### What is a Java Bean ?

**Ans :** Java Bean is a simple java helper class, used to transfer data between classes or applications. It never acts as a main class but like postman or delivery boy between two classes. It doesn’t contain any logic. In order to pass more than 3 values from one class to another or from one project to another, we should take support of Java beans. If less than 3 values, we should use method arguments to transfer the data. However a good design in java doesn’t recommend us to have more than 3 method parameters. You can check java library(java doc), you will not find more than 3 parameters in a methods, very rarely it may have 4 but not more than that.

There are some standard guidelines to develop a Java Bean class :

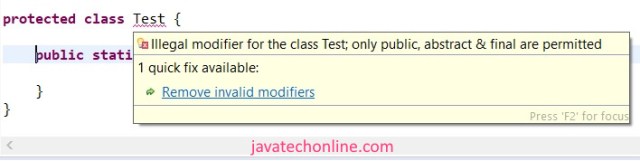
1. We must declare Java Bean as a public class.  
2. It is a recommendation to implement java.io.Serializable (used to sent data across network as we can easily send serializable data over the network). It is optional if you don’t want to send data over network.  
3. All member variables(bean properties) should be private & non-static (if we want to send data of 20 employees, 20 values will be sent. if its [static](https://javatechonline.com/static-keyword-in-java/) only one value will be sent, so declaring non-static becomes mandatory to get full benefit of Java Bean)  
4. Also Every bean property should have one setter and one getter method (Accessor Methods).

**Why are the Java primitive data types not objects? Why is Java not a pure Object-Oriented programming Language?**

Objects always consume more memory than primitives. If we can solve our purpose without creating more number of objects, the performance will always be better. Java Wrapper Classes were introduced after primitives to make a programmer feel Java as pure Object Oriented Programming language. But still because of existence of primitives only Java is not a pure Object Oriented Programming Language.

**What is the difference between Access Specifier & Access Modifiers in java ? Do we have Access Specifiers in java?**

**Ans:**First of all we should know that there is no term called Access Specifiers in java. Other languages like C, C++ etc. may have defined this term but java doesn’t. Some of the books & websites refer private, public, protected, default as Access Specifier but Java compiler considers them as Access Modifiers. If we declare any class as private or protected we can check the compiler error by mouse hover on it. Compiler will show “Illegal modifier for the class Test; only public, abstract & final are permitted” as shown below in the snippet. Therefore, all the related keywords like static, abstract, final, synchronized etc. including private, public, protected are Access modifiers in java.

No Access specifiers in Java

--

**What all access modifiers are allowed in a top level class & in an inner class?**

**Ans :** Most of the times we work with public classes. So there is a high probability that you find this question in the list of Java Interview Questions of interviewer. Generally they target the area that people ignore but these concepts come under the category of basic concepts. Let’s see below the list of allowed access modifiers to each one.

**Top level classes ⇒ default, public, abstract, final, strictfp**

**Inner classes       ⇒ default, public, abstract, final, strictfp, private, protected, static**

**What is strictfp modifier? Where is it applied ?**

**Ans:**strictfp modifier was introduced in jdk 1.2. Full form of strictfp is strict floating point. It is applicable for classes, methods but not for variables. The arithmetic of floating point numbers generally varies from platform to platform ie. platform dependent. Suppose we have three platforms where we want to test value of  an arithmetic 20.0/3 as shown below.

1) for Windows => value may be 6.6666666665  
2) for Linux => value may be 6.6666666  
3) for MAC => value may be 6.666666667

If we want platform independent results for floating point arithmetic, we should use strictfp modifier by declaring it before method. When a method is declared as strictfp then all floating point calculations in that method will follow [IEEE 754 standard](https://en.wikipedia.org/wiki/IEEE_754) and we will get platform independent result. If a class is declared as strictfp, all concrete methods of the class will follow the IEEE 754 standard in the arithmetic calculations.

**♥ strictfp can only be applied with concrete method. Therefore abstract+strictfp combination is illegal for methods.**

**♥ abstract+strictfp is a legal combination for classes as strictfp is applied only to concrete methods.**

**How many types of variables are possible in java? What all access modifiers are allowed with variables?**

**Java Interview Questions**

* [Core Java](https://javatechonline.com/category/java/core-java/)

* [java](https://javatechonline.com/category/java/)

* [Java Interview Questions](https://javatechonline.com/category/java-interview-questions/)

*by*[*devs5003*](https://javatechonline.com/author/erdsingh24/)*- May 15, 2021*0

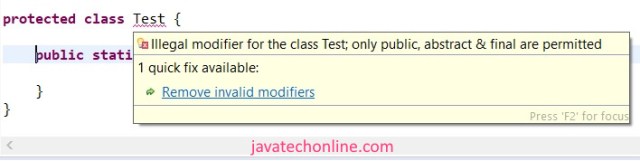
**Error! Filename not specified.**Here in this section we will see Java Interview Questions with answers which will be valuable in any type of Java interview. I am sure that you will find some unique questions that you may rarely see in other sources. Moreover, you will find clear explanation of answers to each question. When you go through the answers you will find that while answering the questions related new terms are also explained. If you need any other question to be added here, don’t hesitate to  put your comment in comment section.

Why are the Java primitive data types not objects? Why is Java not a pure Object-Oriented programming Language?

Objects always consume more memory than primitives. If we can solve our purpose without creating more number of objects, the performance will always be better. Java Wrapper Classes were introduced after primitives to make a programmer feel Java as pure Object Oriented Programming language. But still because of existence of primitives only Java is not a pure Object Oriented Programming Language.

**What is the difference between Access Specifier & Access Modifiers in java ? Do we have Access Specifiers in java?**

**Ans:**First of all we should know that there is no term called Access Specifiers in java. Other languages like C, C++ etc. may have defined this term but java doesn’t. Some of the books & websites refer private, public, protected, default as Access Specifier but Java compiler considers them as Access Modifiers. If we declare any class as private or protected we can check the compiler error by mouse hover on it. Compiler will show “Illegal modifier for the class Test; only public, abstract & final are permitted” as shown below in the snippet. Therefore, all the related keywords like static, abstract, final, synchronized etc. including private, public, protected are Access modifiers in java.

No Access specifiers in Java

**What all access modifiers are allowed in a top level class & in an inner class?**

**Ans :** Most of the times we work with public classes. So there is a high probability that you find this question in the list of Java Interview Questions of interviewer. Generally they target the area that people ignore but these concepts come under the category of basic concepts. Let’s see below the list of allowed access modifiers to each one.

**Top level classes ⇒ default, public, abstract, final, strictfp**

**Inner classes       ⇒ default, public, abstract, final, strictfp, private, protected, static**

**What is strictfp modifier? Where is it applied ?**

**Ans:**strictfp modifier was introduced in jdk 1.2. Full form of strictfp is strict floating point. It is applicable for classes, methods but not for variables. The arithmetic of floating point numbers generally varies from platform to platform ie. platform dependent. Suppose we have three platforms where we want to test value of  an arithmetic 20.0/3 as shown below.

1) for Windows => value may be 6.6666666665  
2) for Linux => value may be 6.6666666  
3) for MAC => value may be 6.666666667

If we want platform independent results for floating point arithmetic, we should use strictfp modifier by declaring it before method. When a method is declared as strictfp then all floating point calculations in that method will follow [IEEE 754 standard](https://en.wikipedia.org/wiki/IEEE_754) and we will get platform independent result. If a class is declared as strictfp, all concrete methods of the class will follow the IEEE 754 standard in the arithmetic calculations.

**♥ strictfp can only be applied with concrete method. Therefore abstract+strictfp combination is illegal for methods.**

**♥ abstract+strictfp is a legal combination for classes as strictfp is applied only to concrete methods.**

**How many types of variables are possible in java? What all access modifiers are allowed with variables?**

**Ans:** Variables are placeholders in java. They are of 4 types in java:

1. Instance Variables (Non-Static Fields)

2. Class Variables (Static Fields)

3. Local Variables

4. Parameters

**How can you decide the jdk version of a compiled .class file?**

**Ans:** Use below command to get Major & Minor version of the file:

**javap -verbose ClassNameWithDirectoryPath | find “version”**

**Is the following main method valid in java ?**

Checking main() Method Validity

public class MainMethodValidity {

synchronized final public strictfp static void main(String... str) {

System.out.println("Am I valid ?");

}

}

**Ans:** Yes this is valid main() method in java. Access modifiers place doesn’t matter. If we want only one thread can access the main method we can use synchronized keyword in it. If we don’t want other class to extend from this class we can declare it as final. In order to know more about strictfp, check the answer of [this question](https://javatechonline.com/java-interview-questions/#What_is_strictfp_modifier_Where_is_it_applied). Starting from JDK 1.5 we can declare varargs as “String… str” in place of ‘String[ ] args’.

**Why can an abstract method not be declared as private ? Can we use private keyword with abstract keyword ?**

An abstract method can reside either in an abstract class or in an interface. The purpose of declaring an abstract method is to force implementing/child classes to provide concrete implementation of it. If we want implementing/child classes to provide concrete implementation of it, they should have access to that abstract method. If we declare the method as private, no other classes can access it as private access modifier is limited to the same class. In this case abstract method will never be implemented. So if we use abstract we want to implement it in one side where in other side using private keyword restricts us to access it. Hence, we can’t use abstract & private together.

**Can an Abstract class contain final method ? What is the benefit of having a final method in an abstract class?**

**Ans:**Yes. An abstract class can contain final method. It is a valid combination, compiler will not complain. When we don’t need the child class to override a method of it’s parent class, we declare the method as final. In general we observe this scenario in Template Method pattern of Java Design patterns. Template method design pattern is to define an algorithm as skeleton of operations and leave the details to be implemented by the child classes. Parent abstract class contain complete implementation a method and used in the algorithm. This method in base class should be restricted so that the sub classes does not override it. So it is declared as final.

**Can we use final & abstract both access modifiers in a method? Can we declare a method as abstract & final both.**

**Ans:**No. It’s illegal contract in java. Abstract will force method to get overridden by child classes whereas final will not allow method to be overridden by child classes. So both are contradictory situation which will not solve the purpose of the method at all. Therefore we can’t declare a method as abstract & final both. These questions on combination of multiple access modifiers are very important. They can be part of Java Interview Questions.

**Is below line a valid class declaration  ?**

**class C implements A extends B {.........}**

**Ans:**No. Its not a valid declaration. The valid declaration is :

**class C extends B implements A {......}**

extends keyword always comes before implements keyword. When we use extends it means we can have methods which are inherited by sub class by default while in case of implements it tells us to override methods forcefully. So we always write first which we already have then others.

**What is the difference between final, finally & finalize() keyword in java ?**

**Ans:** Below is the explanation of all three keywords with complete clarity. Keep your special attention on finally & finalize.

**Final :** Final is a keyword that we use with variable, methods & classes. final variable means it’s value is constant and we can’t reassign a value for that variable. A final method can’t be overridden by a child class. Hence, we can’t create a sub class of a final class.

**Finally :** It is used in context of exception handling with try-catch block to release the resources that we opened in the try block.

**Finalize :** It is a method defined in Object class. Garbage collector calls this method to perform clean up activities before destroying the object from the memory.

**\*♥** Finally meant for cleanup activities related to try block whereas finalize() meant for cleanup activities related to object.

#### **How to set maximum & minimum heap sizes ?**

**1.  By using command prompt execution of program :**

java -Xmx512m JavaProgramFileName enter  
java -Xms64m JavaProgramFileName enter  
Where Xmx indicates Maximum Memory & Xms indicates Minimum Memory. Heap memory is finite memory but based on our requirement we can set max & min heap sizes.

**2**. **By Setting ‘JAVA\_OPTS’ as a system variable**

JAVA\_OPTS=”-Xms256m -Xmx512m”  
After that in a command prompt run the following command:  
SET JAVA\_OPTS=”-Xms256m -Xmx512m”  
This setting indicates  
allocating minimum 256MBs of heap  
allocating maximum 512MBs of heap

**What are the different useful commands to manage jar files?**

We have different commands to create jar file based on some parameters.

1) **jar  -cvf  test.jar Test.class**  ⇒ To create jar file of only one class

2) **jar  -cvf  test.jar** **Test1.class Test2.class Test3.class**  ⇒  To create jar file of multiple classes

3) **jar  -cvf  test.jar \*.class**  ⇒  To create jar file of all the classes in current path

4) **jar  -cvf  test.jar \*.\*** ⇒  To create jar file of all type of files in current path

5) **jar  -xvf  test.jar** ⇒  To extract(unzip) a jar file

6) **jar  -tvf   test.jar** ⇒  To display name of all the files inside the jar

**What is covariant return type in java?**

**Ans:** The concept of covariant return type was introduced in jdk 1.5. Before jdk 1.5 in method overriding the overriding method must have the same return type as it’s parent class method has. In this case, overriding method was said to be invariant with respect to return type. But jdk 1.5 onward, the overriding method’s return type can be the sub-type of it’s parent class method’s return type. This is called **covariant return type.** The **covariant return type** always works only for non-primitive return types. In order to understand with an example, [click here](https://javatechonline.com/java-interview-questions/#Is_the_following_code_valid_override).

**What is difference between ClassNotFoundException & NoClassDefFoundError ?**

**Ans: ClassNotFoundException** is a checked Exception. We face this when we try to load class at runtime using Class.forName or the class is not found in the class path. Most of the time, this exception occurs when you try to run an application without updating the class path with required JAR files.

**NoClassDefFoundError** is an unchecked exception. We face this when we create a hard coded class by using new operator. It occurs when required class definition is missing at runtime while it was available at compile time. For example, if we have a method call from a class or accessing any static member of a Class and that class is not available during run-time then JVM will throw NoClassDefFoundError.

**What is ‘stream’ in Java ? What was the purpose of introducing it ? How does it differ from a collection?**

**Ans:** java.util.stream is a package introduced in jdk 1.8. It has several interfaces & classes to work on streams(flow of data). Classes under stream package introduced to support functional-style operations on the stream of elements. However, please note that Stream is not a collection rather it is a technique to make processing of collection’s data easy by supporting functional-style operations. Furthermore, Streams differs from a collection in several ways.

♠ A stream is not a data structure that stores elements.  
♦ An operation on a stream produces a result, but does not modify its source.  
♥ While collections have a finite size, streams need not.  
♣ The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

**What is a referenced variable in Java ? How many types of reference variables are possible?**

**Ans:** When we create a variable by using Objects (as a Datatype), we call them referenced variables. However, we create them by using Array, Class, Interface or Enum. Moreover, referenced variables store reference of the object while primitive variables store the values directly. Types of possible referenced variables are : Local Static Non-Static Final Volatile Transient

**Abstraction:**

Abstraction is a process of hiding the implementation details from the user, and only providing the functionality will be providing to the user.

In java, abstraction is achieved using abstract classes and interfaces.

Abstract class is a collection of abstract methods and concrete methods (Non abstract);

Abstract class cannot be instantiated.

In Abstract class cannot create the constructor.

**package** com.oops;

**public** **abstract** **class** AbstaractEx {

**public** **int** x;

**void** m1() {

System.***out***.println("Abstract M1 method");

}

**abstract** **void** m2();

}

**package** com.oops;

**public** **class** AbstractClassImpl **extends** AbstaractEx {

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

//AbstaractEx AbstaractEx = new AbstaractEx(); // cannot instantiate abstract class

AbstractClassImpl abstractClassImpl = **new** AbstractClassImpl();

abstractClassImpl.m1();

abstractClassImpl.m2();

abstractClassImpl.x =100;

System.***out***.println("printing abstarct class x values : "+abstractClassImpl.x);

}

@Override

**void** m2() {

System.***out***.println("AbstractClassImpl - M2 Method");

}

}

**Interface:**

Interface is a collection of abstract methods.

Interface cannot be instantiated.

In order to access the members of interface we need to inherit the interface in to the class using implements keyword.

In the subclass we have to override all the abstract methods

If the subclass is overriding all the methods of an interface then it is called as a implementation class.

A class can implement any number of interfaces.

Cannot create the constructors in interfaces

Ex:

**package** com.oops;

**public** **interface** MyInterface {

**public** **int** ***i***=10; // by default varibles are static and final

**public** **void** m1(); // default all methods having the abstract keyword

}

**package** com.oops;

**public** **interface** SecondInterface {

**public** String ***var*** = "Welcome";

**public** **void** m2();

}

Implementation class

**package** com.oops;

**public** **class** MyInterfaceImpl **implements** MyInterface,SecondInterface {

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

MyInterfaceImpl myInterface = **new** MyInterfaceImpl();

myInterface.m1();

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

myInterface.m2();

System.***out***.println(myInterface.***var***);

}

@Override

**public** **void** m1() {

System.***out***.println("M1 Method Executed");

}

@Override

**public** **void** m2() {

System.***out***.println("M2 Method Executed");

}

}

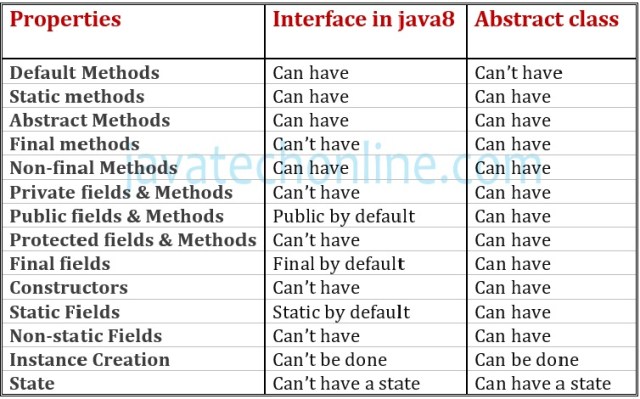
Class -🡪 class --🡪 extends (single class)

Class -🡪interface -🡪 implements (any number of interfaces)

Interface -🡪interface -🡪 extends(any number of interfaces)

### What is difference between Abstract class & Interface in Java 8?

**Ans:** After learning Interfaces in context of Java 8, one doubt may arise in your mind that what is difference between interface & abstract classes after introduction of default & static methods. Let’s clear the doubt from the table below:



### How many ways are there to implement Functional Interface in java?

**Ans :** There are five ways to implement [Functional Interface](https://javatechonline.com/functional-interface-java8/) in Java.

1. By Explicit Outer class  
2. By Explicit Inner class  
3. By Anonymous Inner class  
4. By [Lambda expression](https://javatechonline.com/the-lambda-%CE%BB-expression/#How_to_write_Lambda_Expressions)  
5. By [Method Reference](https://javatechonline.com/method-reference-in-java-8/)

### Jdk 1.0 to J2SE 1.4 :=========

Jdk 1.0 ( Jan23, 1996 ) : Basic language features with APIs  
Jdk 1.1  (Feb 19, 1997 ) : Inner classes, JDBC, Java Beans  
J2SE 1.2 (Dec8, 1998 ) : strictfp keyword, Collection API, Reflection API, JIT compiler  
J2SE 1.3 (May 8, 2000) : Hotspot JVM, JNDI API  
J2SE 1.4 (Feb 6, 2002)  : Assert statements, Exception Chaining,Regular Expressions, XML Parsers(DOM,SAX)

### Java 5.0 :(Sep 30, 2004): ===========

• static imports  
• @Override annotation  
• [Covariant Returns](https://javatechonline.com/java-interview-questions/#What_is_covariant_return_type_in_java)  
• Auto-boxing and auto-Unboxing  
• Var-arg length methods  
• Enhanced for loop  
• Generic Types  
• enum (Enumerated types)  
• Annotations -Meta data  
• C style formatted Input and Output  
• Scanner class  
• Queue type collection classes  
• java.util.concurrency package

### Java 6 :(Dec 11, 2006): ===========

• Console  
• NavigableSet  
• NavigableMap

### Java 7 :(Jul 28, 2011): ===========

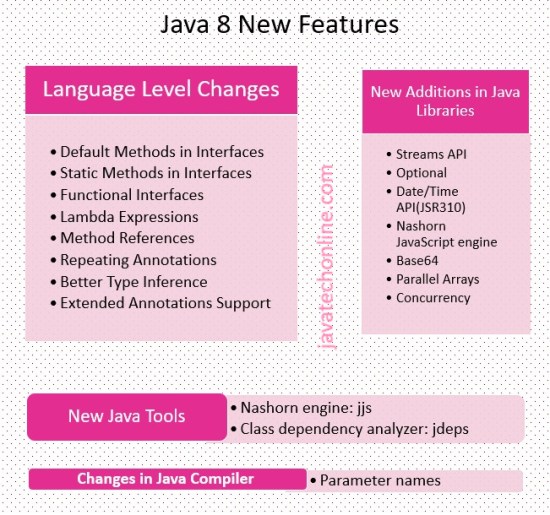
• class execution change with respect to Main Method  
• Binary literals  
• Numeric Literals with underscore  
• switch with String type argument  
• try-with-resource statement  
• catch block with multiple exception handling syntax  
• Var-arg bug fix in overloading & overriding  
• Improved Auto-boxing casting  
• Type inference for Generics

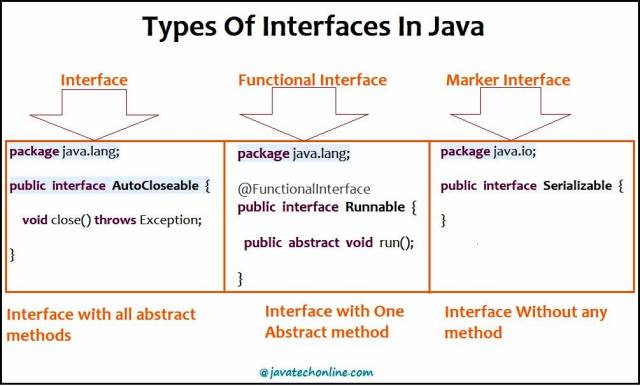
So this is the history of ‘Java Features Before Java8’ where we have seen the features of versions before java 8..

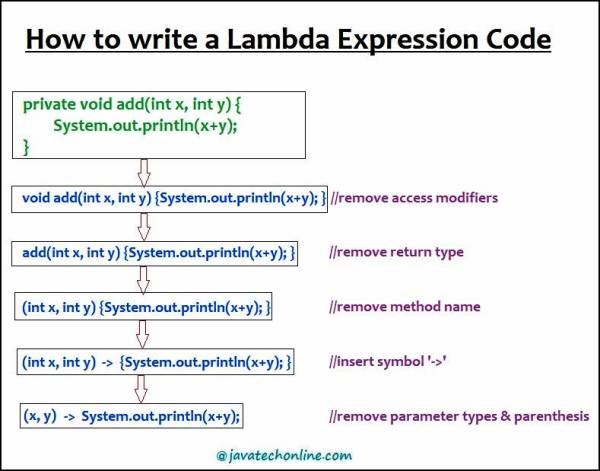
Serialization is a mechanism of writing the state of the objects in to byte stream.

The reverse operation of serialization is called deserialization.(byte stream to actual object).

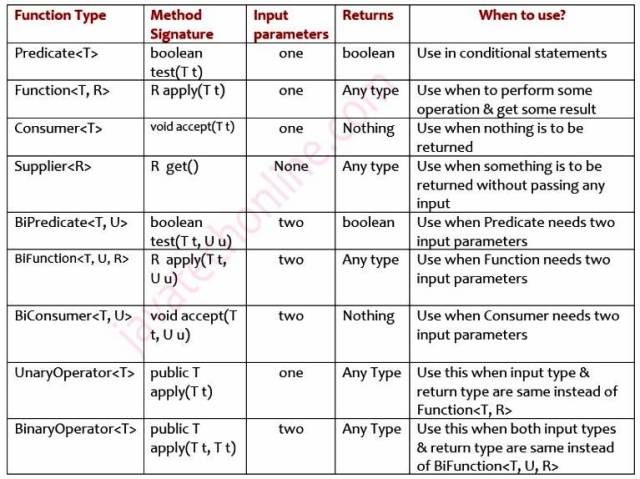
It is mainly used to travel objects state on the network(known as marshaling).

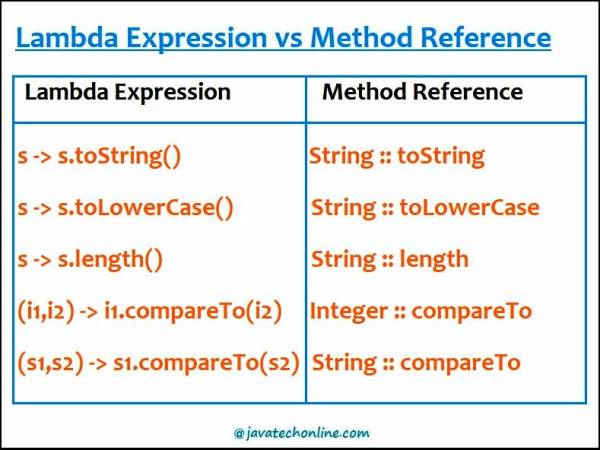






Predefined functional interfaces





## What is a Lambda (λ) Expression

Lambda Expression is an anonymous (nameless) function. In other words, the function which doesn’t have the name, return type and access modifiers. Lambda Expression is also known as anonymous functions or closures.

We use lambda expressions when we provide implementation of [Functional Interfaces](https://javatechonline.com/functional-interface-java8/)which is also introduced as a new feature of java 8. Lambda expressions can also be used in a method as an argument which is one of the important feature of functional programming as the primary focus of Java 8. It provides a clear and concise way to represent one method interface using an expression only. Also, it is very useful in collection & Streams API. It should not be an exaggeration to say that Lambda is the biggest feature of Java 8.

## Benefits Of Using Lambda Expressions

1) We can write more readable, maintainable & concise code using Lambda expressions.  
2) Also, we can incorporate functional programming capabilities in java language with Lambda Expressions.  
3) We can use Lambda expressions in place of Inner classes to reduce the complexity of code accordingly.  
4) Even we can pass a lambda expression as an argument to a method.  
5) Additionally, we can provide lambda expression in place of an object.

### Example of a Thread class with Anonymous Inner class

public class ThreadTest {

public static void main(String[] args) {

Thread t = new Thread(new Runnable() {

public void run() {

for (int i = 0; i < 10; i++) {

System.out.println("Child thread is running");

}

}

});

t.start();

for (int i = 0; i < 10; i++) {

System.out.println("Parent thread is running");

}

}

}

### Converted Example using Lambda Expression

public class ThreadTest {

public static void main(String[] args) {

Thread t = new Thread(() -> {

for (int i = 0; i < 10; i++) {

System.out.println("Child thread is running");

}

});

t.start();

for (int i = 0; i < 10; i++) {

System.out.println("Parent thread is running");

}

}

}

Best Regards  
Ramesh Raju.S

# How many Ways to Create an Object in Java

There are five different ways to create an object in Java:

* Java new Operator
* Java Class.newInstance() method
* Java newInstance() method of constructor
* Java Object.clone() method
* Java Object Serialization and Deserialization

## **1) Java new Operator**

This is the most popular way to create an object in Java. A new operator is also followed by a call to constructor which initializes the new object. While we create an object it occupies space in the heap.

**Syntax**

1. class\_name object\_name = **new** class\_name()

**Example of Java new Operator**

8.4M

134

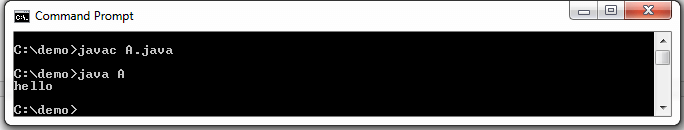
Hello Java Program for Beginners

**Next**

**Stay**

1. **public** **class** A
2. {
3. String str="hello";
4. **public** **static** **void** main(String args[])
5. {
6. A obj=**new** A();  //creating object using new keyword
7. System.out.println(obj.str);
8. }
9. }

**Output:**



## **2) Java Class.newInstance() method**

Java **Class.newInstance()** is the method of Class class. The Class class belongs to **java.lang** package. It creates a new instance of the class represented by this Class object. It returns the newly created instance of the class.

**Syntax**

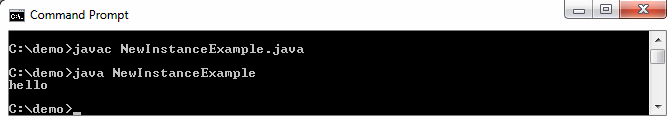
1. **public** T newInstance() **throws** IllegalAcccessException, InstantiationException

It throws **IllegalAccessException** if the class or its nullary constructor is not accessible. It also throws **InstantiationException**, if the Class represents an abstract class, an interface, an array class, or a primitive type.

**Example**

1. **public** **class** NewInstanceExample
2. {
3. String str="hello";
4. **public** **static** **void** main(String args[])
5. {
6. **try**
7. {
8. //creating object of class
9. NewInstanceExample obj= NewInstanceExample.**class**.newInstance();
10. System.out.println(obj.str);
11. }
12. **catch**(Exception e)
13. {
14. e.printStackTrace();
15. }
16. }
17. }

**Output:**



## **3) Java newInstance() method of Constructor class**

Java **Constructor** class also has a **newInstance()** method similar to newInstance() method of Class class. The newInstance() method belongs to **java.lang.reflect.Constructor** class. Both newInstance() method are known as reflective ways to create object. In fact the newInstance() method of Class class internally uses newInstance() method of Constructor class. The method returns a new object created by calling the constructor.

**Syntax**

1. **public** T newInstance(Objects...initargs)

The newInstance() method throws the following Exception:

* **IllegalAccessException:** If the constructor is inaccessible.
* **IllegalArgumentException:** If the actual and formal parameter differ in number.
* **InstantiationException:** If the class constructor represents an abstract class.
* **InvocationTargetException:** If the underlying constructor throws an exception.
* **ExceptionInInitializerError:** If the initialization provoked by this method fails.

**Example**

1. **import** java.lang.reflect.Constructor;
2. **public** **class** NewInstanceExample1
3. {
4. String str="hello";
5. **public** **static** **void** main(String args[])
6. {
7. **try**
8. {
9. Constructor<NewInstanceExample1> obj =NewInstanceExample1.**class**.getConstructor();
10. NewInstanceExample1 obj1 = obj.newInstance();
11. System.out.println(obj1.str);
12. }
13. **catch**(Exception e)
14. {
15. e.printStackTrace();
16. }
17. }
18. }

**Output:**



## **4) Java Object.clone() method**

Java **clone()** method creates a copy of an existing object. It is defined in **Object** class. It returns clone of this instance. The two most important point about clone() method is:

* The **Cloneable** interface must be implement while using clone() method. It is defined in **java.lang** package.
* The **clone()** method must be override with other classes.

When we use clone() method in class, the class must call **super.clone()** to obtain the cloned object reference.

**Syntax**

1. **protected** Object clone() **throws** CloneNotSupportedException

The method throws the **CloneNotSupportedException** if the Object class does not support the Cloneable interface. This exception also throws when subclass that overrides the clone() method indicates that instance cannot be cloned.

**Example**

1. **public** **class** CloneExample **implements** Cloneable
2. {
3. //creates and returns a copy of this object
4. **protected** Object clone() **throws** CloneNotSupportedException
5. {
6. **return** **super**.clone();
7. }
8. String name = "Microprocessor";
9. **public** **static** **void** main(String[] args)
10. {
11. CloneExample obj1 = **new** CloneExample();     //creating object of class
12. **try**
13. {
14. CloneExample obj2 = (CloneExample) obj1.clone();
15. System.out.println(obj1.name);
16. }
17. **catch** (Exception e)
18. {
19. e.printStackTrace();
20. }
21. }
22. }

**Output:**



## **5) Java Object Serialization and Deserialization**

A class must implement **Serializable** interface which belongs to **java.io** package. The Serializable interface does not have any method and field. They add special behavior to the class. Marker interface does not used in Java 8. It is replace by Annotations.

JVM creates a separate space whenever we serialize and deserialize an object. It does not use any constructor to create an object.

### Object Serialization

The **ObjectOutputStream** class is used to serialize an object. The **Serialization** is a process of converting an object into a sequence of bytes.

The **writeObject()** method of **ObjectOutputStream** class serialize an object and write the specified object to the ObjectOutputStram class. The signature of the method is:

1. **public** **final** **void** writeObject(Object obj) **throws** IOException

The method accepts an object as a parameter.

### Object Deserialization

The process of creating an object from sequence of bytes is called object deserialization. The **readObject()** method of **ObjectInputStream** class read an object from the ObjectInputStram class and deserialize it. The signature of the method is:

1. **public** **final** Object readObject() **throws** IOException

The method does not accept any parameter. It returns an object read from the stream. The method throws the following exceptions:

* **ClassNotFoundException:** If the class of serialized is not found.
* **InvalidClassException:** Something is wrong with a class used by serialization.
* **IOException:** Any of the usual Input/Output related exception.
* **OptionalDataException:** If primitive data was found in the stream instead of objects.

**Example**

In the following example we have first serialized the object and then deserialized the object.

1. **import** java.io.\*;
2. **class** Demo **implements** Serializable
3. {
4. **public** **int** i;
5. **public** String s;
6. **public** Demo(**int** i, String s) //default constructor
7. {
8. **this**.i = i;
9. **this**.s = s;
10. }
11. }
12. **public** **class** DeserializationExample
13. {
14. **public** **static** **void** main(String[] args)
15. {
16. Demo object = **new** Demo(8, "javatpoint");
17. String filename = "Demofile.ser";       //specified file name (must have extension .ser)
18. /\*-----------------Serialization----------\*/
19. **try**
20. {
21. FileOutputStream file = **new** FileOutputStream(filename);  //Saving of object in the file
22. ObjectOutputStream out = **new** ObjectOutputStream(file);
23. out.writeObject(object);            //serialize object
24. out.close();                    //closes the ObjectOutputStream
25. file.close();                   //closes the file
26. System.out.println("Object serialized");
27. }
28. **catch**(IOException e)
29. {
30. e.printStackTrace();
31. }
32. Demo obj = **null**;
33. /\*-----------------Deserialization--------\*/
34. **try**
35. {
36. FileInputStream file = **new** FileInputStream(filename); // reading an object from a file
37. ObjectInputStream is = **new** ObjectInputStream(file);
38. obj = (Demo)is.readObject();        //deserialize object
39. is.close();                     //closes the ObjectInputStream
40. file.close();                   //closes the file
41. System.out.println("Object deserialized ");
42. System.out.println("number = " + obj.i);
43. System.out.println("string = " + obj.s);
44. }
45. **catch**(IOException e)
46. {
47. System.out.println("IOException is caught");
48. }
49. **catch**(ClassNotFoundException e)
50. {
51. System.out.println("ClassNotFoundException is caught");
52. }
53. }
54. }

**Output:**



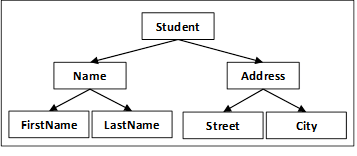
## **Concept of cloning in Java**

In OOPs, copying an object means creating a clone of an existing object. There are many ways to copy an object; two of them are- **copy constructor** and **cloning**. There are two types of cloning in Java:

* Shallow cloning
* Deep cloning

Both deep and shallow copy are types of object cloning. When we talk about an object, we consider it as a single unit which cannot be broken down further.

Suppose we have a Student object. The Student object contains other objects, as in the following figure. The Student object contains Name and Address objects. The Name contains FirstName and LastName objects, and the Address object is composed of a Street and a city object. When we talk about Student, we are talking about the entire network of objects.



A clone of an object is created when we want to modify or move an object while still preserving the original object.

### Shallow Cloning

* Java uses shallow cloning whenever we use the default implementation of the clone() method.
* The shallow cloning of an object creates a clone of the main object but does not copy the inner objects.
* The inner objects are shared between the original object and its copy.

For example, if we want to create a shallow copy of the Student, we should create a second object of Student. But both objects share the same Name and Address. Consider the following Example:

1. **public** **class** Student
2. {
3. **private** Name name ;
4. **private** Address address;
5. **public** Student(Student orgStudent)
6. {
7. **this**.name=orgStudent.name;
8. **this**.address=orgStudent.address;
9. }
10. }

A disadvantage of the shallow copy is that the two objects are not independent. When we modify the Name object of one Student, it modifies the other Students objects too.

In the following example, we have a Student object with a reference variable mba; then we make a copy of MBA, creating a second Student object, mca. If mca tries to moveOut() by modifying his Address object, the mba moves with it.

1. Student mba=**new** Student(**new** Name(...), **new** Address(...));
2. Student mca=**new** Student(mba);
3. mca.moveOut(**new** Street(...), **new** City(...));

It is because mba and mca objects shares the same Address object. If we change the Address in one object, it modifies both.

### Deep Cloning

* Deep cloning is a fully independent copy of an object.
* So for the deep copy, we need to ensure all the member class also implement the **Cloneable** interface and override the clone() method of the Object class.

When we modify the Address object of one Student object, it does not modify the other Student object. In the following code we can see that we are not only using a copy constructor on Student object, but we are also using copy constructor on the inner objects as well.

1. **public** **class** Student
2. {
3. **private** Name name ;
4. **private** Address address;
5. **public** Student(Student otherStudent)
6. {
7. **this**.name=**new** Name(otherStudent.name);
8. **this**.address=**new** Address(otherStudent.address);
9. }
10. }

To create a deep clone, we need to keep copying all the Student object nested elements, until there are only primitive types and Immutable left.

1. **public** **class** Street
2. {
3. **private** String name;        //instance variable
4. **private** **int** number;
5. **public** Street(Street otherStreet)
6. {
7. **this**.name=otherStreet.name;
8. **this**.number=otherStreet.number;
9. }

# Recursion in Java

Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.

It makes the code compact but complex to understand.

**Syntax:**

1. returntype methodname(){
2. //code to be executed
3. methodname();//calling same method
4. }

## **Java Recursion Example 1: Infinite times**

1. **public** **class** RecursionExample1 {
2. **static** **void** p(){
3. System.out.println("hello");
4. p();
5. }
7. **public** **static** **void** main(String[] args) {
8. p();
9. }
10. }

Output:

C++ vs Java

hello

hello

...

java.lang.StackOverflowError

## **Java Recursion Example 2: Finite times**

1. **public** **class** RecursionExample2 {
2. **static** **int** count=0;
3. **static** **void** p(){
4. count++;
5. **if**(count<=5){
6. System.out.println("hello "+count);
7. p();
8. }
9. }
10. **public** **static** **void** main(String[] args) {
11. p();
12. }
13. }

Output:

hello 1

hello 2

hello 3

hello 4

hello 5

## **Java Recursion Example 3: Factorial Number**

1. **public** **class** RecursionExample3 {
2. **static** **int** factorial(**int** n){
3. **if** (n == 1)
4. **return** 1;
5. **else**
6. **return**(n \* factorial(n-1));
7. }
9. **public** **static** **void** main(String[] args) {
10. System.out.println("Factorial of 5 is: "+factorial(5));
11. }
12. }

Output:

Factorial of 5 is: 120

**Working of above program:**

factorial(5)

factorial(4)

factorial(3)

factorial(2)

factorial(1)

return 1

return 2\*1 = 2

return 3\*2 = 6

return 4\*6 = 24

return 5\*24 = 120

## **Java Recursion Example 4: Fibonacci Series**

1. **public** **class** RecursionExample4 {
2. **static** **int** n1=0,n2=1,n3=0;
3. **static** **void** printFibo(**int** count){
4. **if**(count>0){
5. n3 = n1 + n2;
6. n1 = n2;
7. n2 = n3;
8. System.out.print(" "+n3);
9. printFibo(count-1);
10. }
11. }
13. **public** **static** **void** main(String[] args) {
14. **int** count=15;
15. System.out.print(n1+" "+n2);//printing 0 and 1
16. printFibo(count-2);//n-2 because 2 numbers are already printed
17. }
18. }

Output:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

JAVA 1.8

Java is an object-oriented language. By introducing lambdas in Java 8, the authors of Java tried to add elements of functional programming in Java. Now you might be wondering what the difference between object-oriented programming and functional programming is?

In **object-oriented programming**, objects and classes are the main entities. If we create a function then it should exist within a class. A function has no meaning outside the scope of the class object.

In **functional programming**, functions can exist outside the scope of an object. We can assign them to a reference variable and we can also pass them to other methods as a parameter.

A **lambda expression** is just an anonymous function, i.e., a function with no name and that is not bound to an identifier. We can pass it to other methods as parameters, therefore, using the power of functional programming in Java.

## How to write a lambda expression[**#**](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/JP2PrpzzjoK#How-to-write-a-lambda-expression)

It might be difficult to understand what lambda is and how to write a lambda without looking at an example. So, let’s look at an example first, and then we will revisit what we just read.

In the below example, we have a functional interface called Greeting. There are two classes HindiGreeting and EnglishGreeting that implement this interface.

1

2

3

4

5

@FunctionalInterface

public interface Greeting {

    void greet();

}





1

2

3

4

5

6

7

8

public class HindiGreeting implements Greeting {

    // Overriding the greet() method from Greeting interface.

    @Override

    public void greet() {

        System.out.println("Namaste");

    }

}





1

2

3

4

5

6

7

8

public class EnglishGreeting implements Greeting {

    // Overriding the greet() method from Greeting interface.

    @Override

    public void greet() {

        System.out.println("Good Morning");

    }

}





Here, we have another class called WellWisher. This class has a method called wish(Greeting greeting) which takes Greeting as a parameter and based on the type of object passed, prints the greeting.

WellWisher.java

EnglishGreeting.java

HindiGreeting.java

Greeting.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

public class WellWisher {

    public static void wish(Greeting greeting) {

        greeting.greet();

    }

    public static void main(String args[]) {

        Greeting hindiGreeting = new HindiGreeting();

        wish(hindiGreeting);  // Passing an object of HindiGreeting.

        Greeting englishGreeting = new EnglishGreeting();

        wish(englishGreeting);  // Passing an object of EnglishGreeting.

    }

}





Run

Save

Reset

When we run the WellWisher class, we get the below output:

Namaste  
Good Morning

This is a classic object-oriented programming example. Now, what if we want our WellWisher class to greet in all the languages available?

Do we need to create a class for each language, e.g., SpanishGreeting, FrenchGreeting, etc?

Isn’t it possible that we don’t create any class and just send a function to the wish(Greeting greeting) method?

Our wish(Greeting greeting) method will directly execute the function that is provided to it and print the greeting.

This is possible through anonymous classes. We will quickly see how this can be done through an anonymous class, and then jump straight back into lambdas.

In the below example, we will change the WellWisher class to use an anonymous class.

WellWisher.java

Greeting.java

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

public class WellWisher {

    public static void wish(Greeting greeting) {

        greeting.greet();

    }

    public static void main(String args[]) {

        // We are passing an anonymous class object to the wish method.

        wish(new Greeting() {

            @Override

            public void greet() {

                System.out.println("Namaste");

            }

        });

    }

}





Run

Save

Reset

In the above example, we don’t need to create a class for each language. We can use an anonymous class, and that does the trick for us in the example above. However, don’t you think that this code is still cumbersome? Although the class is anonymous, we are still creating a class.

To make our code less cumbersome, let’s remove all the unnecessary code step-by-step and create our first lambda expression.

**Step 1** -> The compiler knows that the wish(Greeting greeting) method takes in a parameter of type Greeting. So, we don’t need to specifically create an anonymous class of type greeting.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

public class WellWisher {

    public static void wish(Greeting greeting) {

        greeting.greet();

    }

    public static void main(String args[]) {

        wish(

            public void greet() {

                System.out.println("Namaste");

            }

            );

    }

}





**Step 2** -> We know that the Greeting interface has only one method. So, we don’t need to provide the method name. We are only concerned with the method body.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

public class WellWisher {

    public static void wish(Greeting greeting) {

        greeting.greet();

    }

    public static void main(String args[]) {

        wish(

            public void () {

                System.out.println("Namaste");

            }

            );

    }

}





**Step 3** -> The compiler can understand that the body does not return anything. So, mentioning the return type is redundant. We can also remove the public declaration.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

public class WellWisher {

    public static void wish(Greeting greeting) {

        greeting.greet();

    }

    public static void main(String args[]) {

        wish(

                () -> {

                    System.out.println("Namaste");

                }

        );

    }

}





Please note that we add a -> between the empty brackets and the method body. This is how a lambda expression is declared.

There still is one more improvement we can make. Since the method body contains only a single line, the curly braces are also unnecessary.

WellWisher.java

Greeting.java

1

2

3

4

5

6

7

8

9

10

11

12

public class WellWisher {

    public static void wish(Greeting greeting) {

        greeting.greet();

    }

    // Passing a lambda expression to wish method.

    public static void main(String args[]) {

        wish( () -> System.out.println("Namaste") );

    }

}





Run

Save

Reset

Congratulations!! You have written your first lambda. This is how simple it is to write a lambda expression.

To recap, when we write a lambda expression, we are basically sending a function as a method parameter, and it is directly getting executed.

## Compare the objects program, Comparator example using anonymous class[**#**](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/7XVXj0xP6xj#Comparator-example-using-anonymous-class)

public class Person {

    private String name;

    private int age;

    private String country;

    public Person(String name, int age, String country) {

        this.name = name;

        this.age = age;

        this.country = country;

    }

    public String getName() {

        return name;

    }

    public int getAge() {

        return age;

    }

    public String getCountry() {

        return country;

    }

}

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

public class PersonService {

    public static List<Person> getPersons(List<Person> persons){

        // Created an anonymous Comparator, which sorts the Person object on the basis of Person name.

        Collections.sort(persons, new Comparator<Person>() {

            @Override

            public int compare(Person p1, Person p2) {

                return p1.getName().compareTo(p2.getName());

            }

        });

        return persons;

    }

}

Main

import java.util.ArrayList;

import java.util.List;

public class PersonMain {

    public static void main(String args[]){

        List<Person> persons = new ArrayList<>();

        persons.add(new Person("John" , 23 , "USA"));

        persons.add(new Person("Carl" , 23 , "Australia"));

        persons.add(new Person("Amit" , 23 , "India"));

        persons.add(new Person("Vikram" , 23 , "Bhutan"));

        persons.add(new Person("Kane" , 23 , "Brazil"));

        // Calling getPerson() method which will return the List of Person in sorted order.

        List<Person> sortedPersons = PersonService.getPersons(persons);

        System.out.println("Persons after sorting");

        // Printing the name of each person.

        for(Person person : sortedPersons){

            System.out.println("Person Name : " + person.getName());

        }

    }

}

## Comparator example using a lambda expression[**#**](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/7XVXj0xP6xj#Comparator-example-using-a-lambda-expression)

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

public class PersonService {

    public static List<Person> getPersons(List<Person> persons) {

        // Instead of creating an anonymous class, we have provided a lambda expression.

        Collections.sort(persons, (p1, p2) -> p1.getName().compareTo(p2.getName()));

        return persons;

    }

}

## What are default methods?[**#**](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/R888OpmM5WO#What-are-default-methods?)

Before Java 8, we could only declare abstract methods in an interface. However, Java 8 introduced the concept of default methods. **Default methods are methods that can have a body**. The most important use of default methods in interfaces is to provide additional functionality to a given type without breaking down the implementing classes.

Before Java 8, if a new method was introduced in an interface then all the implementing classes used to break. We would need to provide the implementation of that method in all the implementing classes.

However, sometimes methods have only single implementation and there is no need to provide their implementation in each class. In that case, we can declare that method as a default in the interface and provide its implementation in the interface itself.

## Syntax of default methods[**#**](https://www.educative.io/courses/java-8-lambdas-stream-api-beyond/R888OpmM5WO#Syntax-of-default-methods)

Let’s understand the syntax of default methods through an example. Here, we have an interface with one abstract and one default method:

1

2

3

4

5

6

7

8

public interface Vehicle {

    void cleanVehicle();

    default void startVehicle() {

       ·System.out.println("Vehicle·is·starting");

    }

}





Now we will create a class which implements the vehicle interface.

Car.java

Vehicle.java

1

2

3

4

5

6

7

8

9

10

11

12

13

public class Car implements Vehicle {

    @Override

    public void cleanVehicle() {

        System.out.println("Cleaning the vehicle");

    }

    public static void main(String args[]){

        Car car = new Car();

        car.cleanVehicle();

        car.startVehicle();

    }

}





Run

Save

Reset