Basic Java

Q1. Method vs Function.

Function is a concept of procedural and functional programming language. A function is independent of any object.

But method is a function which is always associated to a class. Therefore, a method is limited to operating on data that is within that class. Hence method is just an object-oriented word for a function.

Java is object oriented; you cannot have Java code to run without classes. Hence Java consists of methods only.

Parameters vs Arguments:

Parameters are fields that serve as variable names inside of a method signature.

Arguments are the values passed to the method when it is called.

Q2. Compiled languages vs Interpreted languages.

Interpreted languages:

In an interpreted language, the source code is not directly executed by the target machine. Instead, an intermediate program, aka the interpreter, reads and executes the code.

You trade faster speed of development for higher execution costs. Examples are Python, JavaScript, etc.

Compiled languages:

They use compilers to compile the code (into native machine code) before the processor can attempt to execute it. Results into the generation of an executable file that can be run by a computer without any extra dependencies. But you will need to rebuild the executable file every time you make changes to the code.

Here you trade higher development time for faster execution. Examples of purely compiled languages are C, C++.

Note: Java is both a compiled and interpreted programming language. A .java file is first compiled by Java Compiler into .class file and later interpreted by JIT into native machine code.

Q3. Scripting language vs Programming language.  
  
The basic difference between a scripting language and a programming language is that scripting languages do not need an additional step of compilation and rather they are interpreted, whereas programming languages are compiled and hence need a compilation step to convert the high-level language to machine code.

All the scripting languages are programming languages, but all programming languages are not scripting languages.

Q4. Can we have a static constructor in Java?

In Java, a constructor is not allowed to be abstract, final, static or native. So, there is no static constructor in Java.

Since each constructor is being called by its subclass during creation of the object of its subclass, so if you mark constructor as static the subclass will not be able to access the constructor of its parent class because it is marked static and thus belong to the class only. This will violate the whole purpose of inheritance concept and that is reason why a constructor cannot be static.

An alternative to static constructor is a static block, which can be used to initialize all static instance variables in a class.

Q5. Is Java a pure object-oriented programming language?

No. Any Object-oriented language must follow the following principals:

1. Encapsulation/Data Hiding
2. Inheritance
3. Polymorphism
4. Abstraction
5. All predefined types are objects
6. All user defined types are objects
7. All operations performed on objects must be only through methods exposed at the objects.

Java supports property 1, 2, 3, 4 and 6 but fails to support property 5 and 7 given above.

Smalltalk is a “pure” object-oriented programming language unlike Java and C++ as there is no difference between values which are objects and values which are primitive types. In Smalltalk, primitive values such as integers, booleans and characters are also objects. In Java, we have predefined types as non-objects (primitive types).

Even using Wrapper classes does not make Java a pure OOP language, as internally it will use the operations like Unboxing and Autoboxing. So, if you create Integer instead of int and do any mathematical operation on it, under the hoods Java is going to use primitive type int only.

Also, unlike top-level classes, inner classes can be static in Java. Such static classes can be used without the need of an object in Java, defying the object-oriented principal.

Also, several OOPs features like multiple inheritance and operator overloading is not supported by Java.

Q6. Is Java interpreted or compiled programming language? (How is Java a platform independent language?)

The Java source code first compiled into a .class file consisting of byte code (intermediate code) using Java compiler, then this byte code runs on the JVM (Java Virtual Machine), which is a software-based interpreter. So, Java is considered as both interpreted and compiled.

Just In Time compiler which is a part of JVM, performs another round of compilation (interpreting phase) of the byte code into native code of the corresponding platform. This native code can now be interpreted by the processor directly.

Working of JIT in detail:

Imagine that you have two methods; method a() and b(). This a() might be executing for 500th time. Hence, the JIT compiler sees that it is getting executed a lot. So what it does is, pre-compile that method to native code and optimize it further for fast execution. So when the method a() is triggered for the 501st time, instead of Interpreter interpreting the method line by line, the native code corresponding to the particular method (which is optimized) get executed. Now say this method get executed for 2000th time; then the JIT compiler performs another round of optimization on the native code.

But the method b() might not be like method a() and it may be executed for like 10 times. Then that method will be interpreted each time it was invoked.

To handle this process, JIT compiler has 4 parts:

1. Hotspot profiler: Detects and profiles the “Hot spots” i.e. frequently executed code snippets.
2. Intermediate code generator: Responsible for generating the intermediate code.
3. Code optimizer: Performing optimization on generated intermediate code.
4. Target code generator: Generates native code.

Q7. Does Java support Operator Overloading?

Java doesn’t support operator overloading except for one instance, where the + operator is used to add numeric types as well as to append strings.

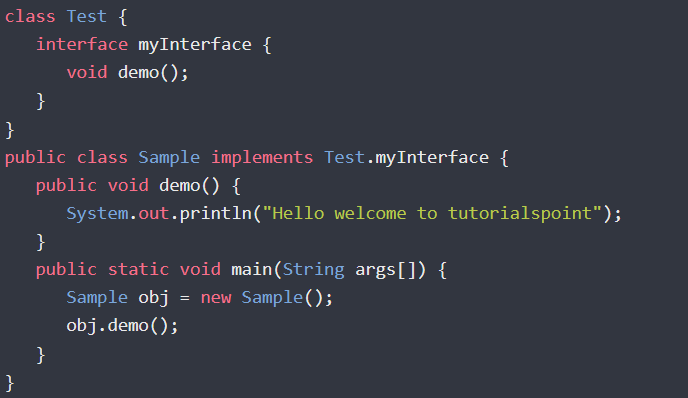
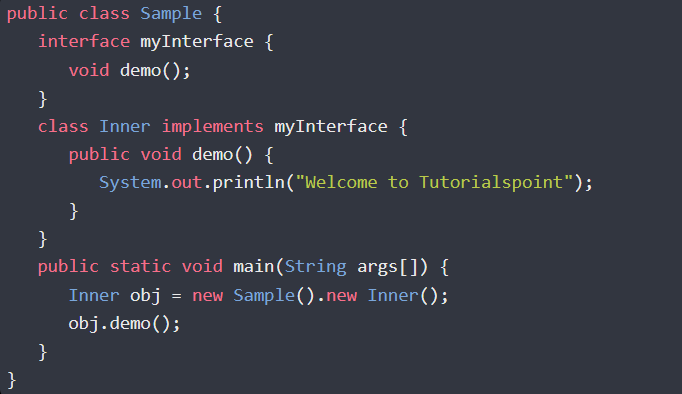
Overloading operators causes faults in the software. Operator overloading causes programmers to become perplexed. Also, compared to other languages, dealing with languages that offer operator overloading has a higher risk of error. Method overloading is a feature of Java. Method Overloading performs the same function as operator overloading while removing the possibility of errors.

Q8. Why do we need Generic Types in programming?

We need generics to implement type-safety and avoid type-casting while using collection classes. Generics was added to java after 1.5 version update.

Q9. Can we define an interface within a Java class?

Yes, you can define an interface inside a class and it is known as a nested interface. But you can’t access a nested interface directly; you need to implement the nested interface using the inner class or you can access it by using the name of the class holding this nested interface.

Q10. Can we define a class inside a Java interface?

Yes, you can define a class inside an interface. In general, if the methods of the interface use this class and if we are not using it anywhere else, we will declare a class within an interface.

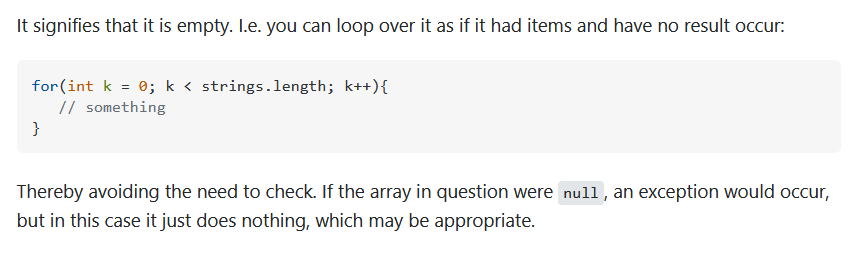


We can also provide default implementation for the methods of interface within this inner class.

Q11. Why does Java allow an array of size 0?

Let’s suppose you have an algorithm where you pop out an element from array and then you trim down the array to get rid of unoccupied positions. And later on, at some point you want to iterate over this array.

Now at some point if all the elements from that array are removed, the iteration part of the program should not throw an error for trying to iterate over an empty array.



Q12. Explain target-type inference in Java.

Type Inference was introduced in Java 5 to complement the introduction of generics. Type inference**is the process of automatically deducing unspecified data types of an expression based on the contextual information.**

Map<String, List<String>> myMap = new HashMap<>();

Q13. Explain static and dynamic binding.

Private, final and static members (methods and variables) use static binding while for virtual methods (In Java methods are virtual by default) binding is done during run time based upon the run time object.

The static binding uses Type information for binding while Dynamic binding uses Objects to resolve to bind.

Overloaded methods are resolved using static binding while overridden methods use dynamic binding, i.e, at run time.

Q14. Explain tight coupling vs loose coupling in Java.

In object-oriented design, Coupling refers to the degree of direct knowledge that one element has of another.

Tight Coupling: When an object creates the object to be used, it is called *tight coupling*.

Loose Coupling: When an object gets the object to be used from external sources, we call it *loose coupling*.

If the classes or methods know less about each other, a loosely coupled structure comes into existence. To achieve loose coupling, one should use abstract classes or interface while performing inheritance.

Example:

class Fruit {

void taste() { sysout(“Fruits have some taste”); }

}

class Mango extends Fruit {

void taste(){ sysout(“Mango is sweet”); }

}

public static void main(String[] args){

Mango m = new Mango();

m.taste();

}

However, there is a problem in the above code. The inheritance has led to the tight coupling of classes. Class Mango knows a lot of stuff about class Fruit. Also, there are fair chances that changes in class Fruit might impact class Mango. Let's modify the above code to understand it.

Suppose we have a requirement to add a parameterized constructor to class Fruit.

class Fruit {

Fruit(String colorOfFruit){ //some logic }

void taste() { sysout(“Fruits have some taste”); }

}

class Mango extends Fruit {

void taste(){ sysout(“Mango is sweet”); }

}

public static void main(String[] args){

Mango m = new Mango();

m.taste();

}

Now after adding a parameterized constructor the same code gives an error in child class Mango (constructor Fruit in class Fruit cannot be applied to given types; class Mango extends Fruit). But we have not touched anything in class Mango. This is a clear disadvantage of tight coupling.

Let's rewrite the above code using an interface.

interface Fruit {

void taste();

}

class Mango implements Fruit {

public void taste(){ sysout(“Mango is sweet”); }

}

class Apple implements Fruit {

Apple(String color){ //some logic }

public void taste(){ sysout(“Apples are sour”); }

}

public static void main(String[] args){

Mango m = new Mango();

m.taste();

}

Now, the parameterized constructor of class Mango is not affecting class Apple (and vice-versa). It is because class Mango and class Apple both are dependent on the abstraction, which is the interface of this case.

Q15. Explain OOPs concepts.

Core OOPs concepts are as follows.

**1.Abstraction:**

It is the process of hiding the internal details of an application from other programs. Abstraction is used to achieve loose coupling, to provide standardization, to layout goals for implementing classes.

We have two types of abstraction: data abstraction and process abstraction.

Data abstraction is making the data members of a class as private and providing the access to those data members via public methods (getters and setters).

Process abstraction is to hide the internal implementation of the different functions involved in a user operation. For example, a user operation like SignUp involves various internal implementation, which is hidden from user.

Abstraction in Java is implemented through interfaces and abstract classes. They provide a contract for the implementation classes.

**2.Encapsulation:**

Encapsulation is the technique used to implement data abstraction. It is used for access restriction to class members and methods using appropriate access modifiers.

**3.Polymorphism:**

Polymorphism is the concept where an object behaves differently in different situations. There are two types of polymorphism - compile time polymorphism and runtime polymorphism. Compile-time polymorphism is achieved by method overloading.

Runtime polymorphism is implemented when we have an “IS-A” relationship between objects. This is also called a method overriding because the subclass has to override the superclass method for runtime polymorphism. If we are working in terms of the superclass, the actual implementation class is decided at runtime.

**4.Inheritance:**