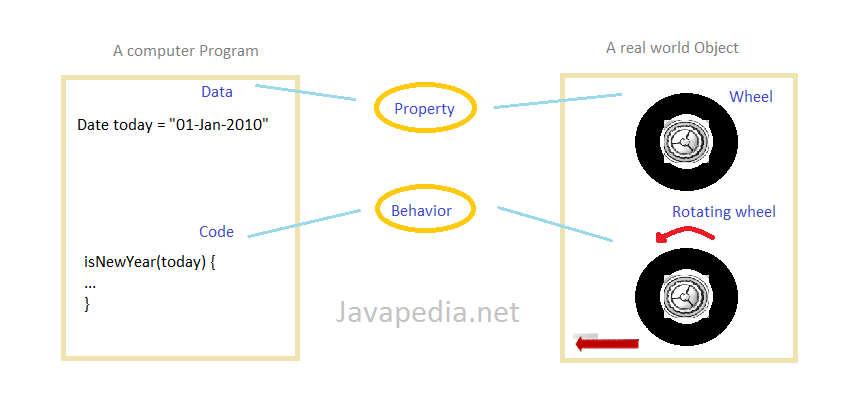
**OOP**

Object Oriented programming is a type of programming paradigm where programs are treated as real-world objects. OOPS correlates a computer program into a real-world object.

A real-world object, for example, a car, has properties and its associated behaviors. Car has wheels as property and its associated behavior is rotate, flat tire etc. Similarly, a computer program is considered to have data (property) and code or logic that manipulate that data (behavior).



**Principles of OOP.**

The below are the principles that attributes to the OOP behavior.

* Inheritance
* Abstraction
* Encapsulation
* Polymorphism
* Aggregation
* Composition
* Association

**Open closed principle**.

In object-oriented programming world, the open-closed principle states that the software components classes, methods etc. should be open for extension but closed for modification. A class can allow its behavior to be extended without modifying its own behavior.

**Difference between object oriented and object-based language.**

Object oriented language supports all the features/principles of OOP whereas object-based languages do not support all the features.

e.g. JavaScript does not support inheritance and polymorphism.

C++, Java are examples of object-oriented and JavaScript and VB are object-based.

**Define class.**

A class is a blueprint/prototype or a template for creating different objects which define its properties and behaviors.

Java class objects exhibit the properties and behaviors defined by its class. A class provides fields and methods that describe the behavior of an object.

The cast in the below picture represents the class and the impression on the sand are the objects.

**Object.**

"Object" refers to an instance of a class where the object can contain any combination of variables, functions.

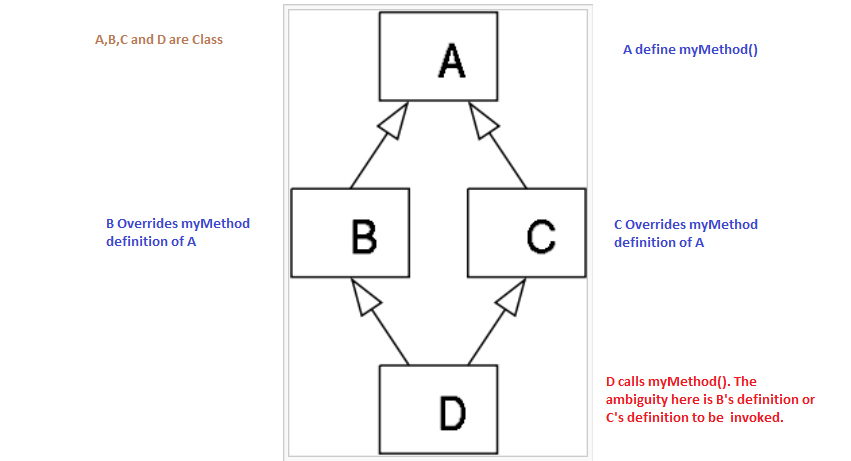
**Define a variable/data.**

A variable holds or stores some data; one variable might be a color of car; another variable might be model.

**Explain Diamond problem**.

The diamond problem (a.k.a deadly diamond of death) refers to an ambiguity that brews due to allowing multiple inheritance.

In Java, multiple inheritance is not allowed for classes and permitted only for interfaces to eliminates this serious issue.



SOLID principles of OOD (Object Oriented Design).

SOLID principles provide the specification for the design of a class to ensure high quality.

There are five principles as each letter in SOLID represents one.

**The Single Responsibility Principle.**

Every class should only serve a single purpose and that responsibility be encapsulated at the same class. For e.g. creating a CAR Class should define the properties that a car would have and along with its functions. It does not have to have Driver Class who drives it.

**The Open Closed Principle.**

The Open-Closed Principle (OCP) states that classes should be open for extension but be closed for modification.

**The Liskov Substitution Principle.**

An extension of the Open Close Principle and ensures that new derived classes(child) are extending the base class(parent) without altering its behavior.

**The Interface Segregation Principle.**

The Interface implementing class should not be forced to implement methods that it doesn’t use. If it forces, then it is not compliant to Interface segregation Principle and this interface referred as fat interface. The Fat interface could be fragmented further to be in align with Interface Segregation principle.

**The Dependency Inversion Principle.**

The principle states:

A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

B. Abstractions should not depend on details. Details should depend on abstractions.

**Why Java does not support operator overloading?**

Operator overloading makes the code less readable and poorly maintainable. To maintain code simplicity, Java doesn't support operator overloading.

**Differences Between Static Binding and Dynamic Binding in Java**.

|  |  |
| --- | --- |
| **Static Binding.** | **Dynamic Binding.** |
| binding that happens at **compile time**. | binding that happens at **run time**. |
| Actual object is not used for binding. | Actual object is used for binding. |
| **Method overloading** is the best example of static binding. | **Method overriding** is the best example of dynamic binding. |
| Private, static and final methods exhibit static binding. Because, they cannot be overridden. | Methods other than private, static and final exhibit dynamic binding. Because, they can be overridden. |
| also called as **early binding** because binding happens during compilation. | also called as **late binding** because binding happens at run time. |

In a class, a method is overloaded, and One form is declared static while another form is non-static. Is that method properly overloaded?

Yes. Compiler checks only method signature to validate if a method is overloaded or not. It does not check static or non-static nature of the method.

**What is Encapsulation in Java?**

Encapsulation in Java is a concept that enforces protecting variables, functions from outside of class, to better manage that piece of code and having least impact or no impact on other parts of a program due to change in protected code.

You can completely encapsulate a member be it a variable or method in Java by using **private keyword**and you can even achieve less encapsulation in Java by using other access modifiers like protected or the public.

Difference between data hiding and data abstraction in OOPS.

**Data hiding**is when the designer specifically decides to limit access to details of an implementation.

**Abstraction** is when you are dealing with an aggregate, for example, a Car is an abstraction of details such as a Motor, Wheels, brake, etc. Abstractions allow us to think of complex things in a simpler way.

**Difference between abstraction and encapsulation in OOPS.**

Abstraction deals with separating interface from implementation. Abstraction in java is achieved by using interface and abstract class. Interface ensures 100% abstraction and abstract class provide 0-100% abstraction.

Encapsulation restricts access to or knowledge of internal structures of an implementation.

**Which of the Java OOPS feature promotes access protection or data hiding?**

**Encapsulation**promotes access protection and data hiding.

**Explain Liskov Substitution Principle (LSP).**

The Liskov Substitution Principle is a concept in Object Oriented Programming that states: **Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.**

**Explain cohesion and coupling in OOD**.

**Cohesion** refers to what the class (or module) can do.**High cohesion is preferred** since it is focused on actions it can perform and limited. Lower cohesion means that the class does a great variety of actions, broad, unfocused.

**Coupling**refers to how related or dependent two classes/modules are toward each other. For low coupled classes, changing in one class should not affect the other.

So as per good Object analysis and design (OAD), **high cohesion and low coupling** is ideal

**What are the differences between composition and inheritance in Java?**

|  |  |
| --- | --- |
| **Composition** | **Inheritance** |
| has-a relationship between classes. | is-a relationship between classes. |
| Composing object holds a reference to composing classes and hence relationship is loosely bound. | Derived object carries the base class definition in itself and hence its tightly bound. |
| Single class objects can be composed within multiple classes. | Single class can only inherit one Class. |
| It’s the relationship between objects. | It’s the relationship between classes. |

**Are static members inherited to sub classes?**

No. Static members cannot be inherited. However super class and the sub class can have static method with same signature.

Super class static member will be hidden at the sub class.

What happens if the parent and the child class have a field with same identifier?

Super class field member will be hidden at the sub class and super class field could be accessed using super keyword.

**Are constructors and initializers also inherited to sub classes?**

No, Constructors and initializers (Static initializers and instance initializers) are not inherited to sub classes. But, they are executed while instantiating a sub class.

**How do you restrict a member of a class from inheriting by its sub classes?**

By declaring that member as a private. Because, private members are not inherited to sub classes.

**How do you implement multiple inheritance in java?**

Using interface, we can implement multiple inheritance in java. Classes in java cannot extend more than one classes, but a class can implement more than one interfaces.

**How Inheritance is implemented in java?**

Inheritance is implemented in JAVA using below two keywords,

* extends.
* implements.

extends inherits between two classes and two interfaces.

implements inheritance between an interface and class.

**Types of inheritance.**

**Single Inheritance.**

One class is extended by only one class.

**Multilevel Inheritance.**

One class is extended by a class and that class in turn is extended by another class thus forming a chain of inheritance.

**Hierarchical Inheritance.**

One class is extended by many classes.

**Hybrid Inheritance.**

It is a combination of above types of inheritance.

**Multiple Inheritance.**

It is a combination of above types of inheritance.

One class extends more than one classes. (Java does not support multiple inheritance.)

Through interfaces, we can implement multiple inheritance in java. As classes in java cannot extend more than one classes, but a class can implement more than one interfaces

**Can we reduce the visibility of the inherited or overridden method?**

No.

**How do you override a private method in java?**

Private methods cannot be overridden as it is not visible outside of the class.

**When to overload a method in Java and when to override it?**

If a class has a better implementation or most appropriate method implementation, then override the method while overloading is performing the same functionality however with different input datatypes/object.

**What is the order of extends and implements keyword on Java class declaration?**

The **extends always precedes** the implements keyword in any Java class declaration.

When the Java compiler compiles a class into bytecode, it must first look to a parent class because the underlying implementation of classes is to point to the bytecode of the parent class - which holds the relevant methods and fields.

**How do you prevent overriding a Java method without using the final modifier?**

Declare that method using private or static keyword to prevent overriding.

**What are the rules of method overriding in Java?**

private, final and static methods cannot be overridden.

The overriding method must have same argument list.

The overriding method must have same return type or covariant return type.

The overriding method cannot reduce the method visibility.

The overriding method must not throw new or broader checked exceptions.

Overriding method can increase access of overridden method.

Use the super keyword to invoke the overridden method from a subclass.

Constructors cannot be overridden.

The synchronized and the strictfp modifier has no effect on the rules of overriding.

**Difference between method overriding and overloading in Java.**

Method overloading deals with the notion of having two or more methods in the same class with the same name but different arguments. Method overriding means having two methods with the same arguments, but different implementations. One of them would exist in the parent class, while another will be in the derived, or child class.

Calls to overloaded methods are realized at compile time. That’s why it is static. Calls to overridden methods are realized at run-time, based on the type on which the method is invoked.

**What is Virtual method/function in Java?**

In Java, all non-static methods are by default virtual functions. Methods that are marked with the keyword final are non-virtual.

These methods are also known as dynamic binding or dynamic dispatch.

**How to prevent using overloading instead of overriding by mistake?**

Always use @override annotation to avoid such mistakes.

**Difference between Inheritance and Encapsulation.**

Inheritance is an object-oriented concept which creates a parent-child relationship. It is one of the ways to reuse the code written for parent class, but it also forms the basis of Polymorphism.

Encapsulation is an object-oriented concept which is used to hide the internal details of a class, for example, HashMap encapsulate how it store elements and calculate hash values.

**What happens when a class implements two interfaces, and both have a method with same name and signature?**

This is a valid scenario. If a type implements two interfaces, and each interface define a method that has identical signature, then in effect there is only one method, and they are not distinguishable.

The below snippet compiles and runs.

**public** **interface** **InterfaceA** {

**void** **method1**();

}

**public** **interface** **InterfaceB** {

**void** **method1**();

}

**public** **class** **ClassImplementing2Interface** **implements** InterfaceA,InterfaceB {

**@Override**

**public** **void** **method1**() {

System.out.println("hello from method1");

}

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

**new** **ClassImplementing2Interface**().method1();

}

}

Output:

hello from method1

**What happens when a class implements two interfaces, and both declare field (variable) with same name?**

The field becomes ambiguous and we get compile time error.

**Can a subclass instance method override a superclass static method?**

No. It results in compilation error at the subclass. The error will be “This instance method cannot override the static method from Parent class”.

**Can a subclass static method hide superclass instance method?**

No. It results in compilation error in the subclass. This static method cannot hide the instance method from Parent

**What is method hiding in Java?**

Static methods cannot be overridden in Java, but if you declare the same static method in subclass then that would hide the method from its superclass. So, if you call that method from subclass then the one in the subclass will get invoked but if you call the same method from superclass then the one in superclass will be invoked. This is known as method hiding in Java.

**Can a superclass access subclass member?**

No. Parent class cannot access the child class members. Super class reference variable cannot see subclass object members.

**Java Identifiers.**

Identifiers are the names of variables, methods, classes, packages and interfaces.

In the below example, MyClass, MyInterface, method\_name, packagename and intVariable are identifiers.

class MyClass {}.

interface MyInterface {}.

void method\_name() {}.

package packagename;

int intvariable;

**Is null a keyword?**

The null is not a keyword. null is a literal as true or false.

**Can a digit form the first character of an identifier?**

No. Only characters are allowed. However, they may be used after the first character.

**Explain var keyword in Java.**

var keyword is introduced in **Java 10** to enhance the Java Language to extend **type inference to declarations of local variables** with initializers.

var map = **new** HashMap<String, Integer>();

var is not a keyword in Java, but rather it is a **reserved type name**. var cannot be used for fields, return types, class names, or interface names.

var can only be used for implicit type inference in the following contexts.

* local variables with initializers.
* indexes in the enhanced for-loop.
* locals declared in a traditional for-loop.

Since var is a reserved type name, it can still be used for package name, method name, and variable name along with its new type-interference role. For example, the following are all examples of valid uses of var in Java.

var map = **new** HashMap<String, Integer>();

var k= **0**;

var var = **10**;

**for** (var i = **0**; i < **100**; i++) { /\* ... \*/ }

**public** **int** **var**() { **return** -**1**; }

**package** var;

**Architectural-neutral.**

Java programs are compiled to generate byte-code which is .class file. This byte-code files are interpreted by the architecture specific JVM to execute that makes java programs run in any type of hardware.

**What is High level language?**

High level languages(HLL) are languages that has less complexity and easy to understand, as the syntax are close to human language and it abstracts the machine language.

**Features of Java. (or) Java buzzwords.**

There are 12 buzzwords.

1. Simple.
2. Object-Oriented.
3. Platform independent.
4. Secured.
5. Robust.
6. Architecture neutral.
7. Portable.
8. Dynamic.
9. Interpreted.
10. High Performance.
11. Multithreaded.
12. Distributed.

**JVM**

     JVM is Java Virtual Machine, it provides the runtime environment in which java bytecode can be executed. JVMs are available for many hardware and software platforms (so JVM is platform dependent).

**JRE**

     JRE is Java Runtime Environment. It is the implementation of JVM.

JRE doesn't have Javac compiler.

**JDK**

   JDK is Java Development Kit. It exists physically and contains JRE and development tools.

**Java is Platform independent.**

JAVA supports WORA which make it run across different platform/operating system.

**Interpreted.**

The Java compiler generates byte-codes, rather than native machine code. To run a Java program, you use the Java interpreter to execute the compiled byte-codes. Java byte-codes provide an architecture-neutral object file format. The code is designed to transport programs efficiently to multiple platforms.

**Three flavors of Java.**

ME (Micro Edition) for mobile,  
SE (Standard Edition) for desktops,   
and EE (Enterprise edition) for enterprise.

**How many types of memory areas are allocated by JVM?**

Many types:

Class(Method) Area,  
Heap,  
Stack,  
Program Counter Register,  
Native Method Stack.

**What is Write Once, Run Anywhere (WORA)?**

WORA refers to the ability of a computer program to run across any operation system or platform.

**What is a Platform?**

Platform refers to any hardware/software environment under which a program runs.

**Is Java a pure/fully object-oriented language?**

Java has primitive data types which are not objects, so Java is considered as not fully object-oriented language. However, Java could be considered as fully object-oriented language as it has wrapper classes for its primitive types.

**Secured**.

Java is secured for developing application due to many of security features that includes, Byte-code verification before execution, Run time security checks, Immutable classes (String for e.g.)

**What is bytecode?**

Bytecode is a highly optimized set of instructions, low level language and non-executable code. The platform specific JVM interprets bytecode to run the program.

**Robust.**

* highly supported language and portable,
* Strongly typed language,
* Memory Management,
* Garbage Collection,
* Exception Handling,
* No pointers.

**Dynamic**.

Java was designed to adapt to an evolving/changing environment. Even after binaries have been released, they can adapt to a changing environment. Java loads in classes as they are needed, even from across the network It defers many decisions (like object layout) to runtime, which solves many of the version problems that C++ has.

**What is class-loader?**

The class-loader, a subsystem of JVM, is used to load classes and interfaces.

**Types of class-loader.**

There are many types of class-loaders e.g.

* Bootstrap,
* Extension,
* System,
* Plugin.

**What is Heap space in Java?**

Java Virtual Machine(JVM) gets some memory from the underlying operating system to run any java program. This memory is referred as Java heap.

Heap in Java located at bottom of address space and move upwards. When an object is created using new operator or by any other means object is allocated memory from Heap and when the object is garbage collected, memory goes back to Heap space in Java and get reused.

**What are the new features introduced in Java 8?**

* Lambda Expressions,
* Interface Default and Static Methods,
* Method Reference,
* Parameters Name,
* Optional feature,
* Streams,
* and Concurrency.

**Difference between EAR, JAR and WAR file in J2EE**.

J2EE defines three types of archives.

* **Java Archive (JAR)** A JAR file encapsulates one or more Java classes, a manifest, and a descriptor. JAR files are the lowest level of archive. JAR files are used in J2EE for packaging EJBs and client-side Java Applications.
* **Web Archive (WAR)** A WAR files are like JAR files, except that they are specifically for web applications made from Servlets, JSPs, and supporting classes. Struts and Spring based Web applications may be archived to a WAR.
* **Enterprise Archive (EAR)** An EAR file contains all the components that make up a J2EE application. A EAR may contain one or more WAR files. EAR files can also contain connector modules packaged as RAR files and Client modules packaged as JAR files.

**Explain memory leak in Java.**

Memory leak in Java refers to a situation where some objects are not used by the application any more, but GC fails to recognize them as unused. As a result, these objects remain in memory indefinitely, reducing the amount of memory available to the application.

**Types of memory leaks in Java.**

OutOfMemoryError is one of the symptom to be diagnosed as it could lead to memory leaks.

Application may crash without issuing OutOfMemoryError.

**How do you diagonise memory leaks in Java?**

* Identify symptoms,
* Enable verbose garbage collection using -verbosegc argument,
* Enable profiling,
* Analyze the trace.

**Difference between Stack and Heap memory in Java.**

The main difference between heap and **stack is that stack memory is used to store local variables and function call** while **heap memory stores objects** in Java.

Each Thread in Java has its own stack whose size can be specified using -Xss JVM parameter, similarly, you can also specify heap size of Java program using JVM option -Xms and -Xmx where -Xms is the starting size of the heap and -Xmx is the maximum size of java heap.

If there is no memory left in the stack for storing function call or local variable, JVM will throw java.lang.StackOverFlowError, while if there is no more heap space for creating an object, JVM will throw java.lang.OutOfMemoryError: Java Heap Space.

Size of stack memory is a lot lesser than the size of heap memory in Java.

Variables stored in stack is only visible to the owner thread while objects created in the heap are visible to all threads. Stack memory is a private memory of a Java Thread while heap memory is shared among all threads.

**Does Java garbage collector clean both heap and stack memory?**

GC sweeps heap memory only. Usually, stack memory is collected automatically when the execution path reaches the end of the scope.

**Why garbage collection is required in Java?**

Garbage collection relieves programmers from the burden of freeing allocated memory.

GC helps ensure program integrity. Garbage collection is an important part of Java's security strategy.

**How GC does not affect Java program's performance?**

Because Java's garbage collector runs in its own thread, in most cases, it will run transparently alongside the execution of a program.

**Different categories of GC algorithm.**

GC algorithms can be divided into 3 categories: sweeping, compacting and copying.

**List the available GC algorithms in Java.**

**The Serial GC**recommended for client-style applications that do not have low pause time requirements.

**The Parallel GC**- use when the throughput matters. The Mostly-Concurrent GC (also known as Concurrent Mark-Sweep GC(CMS)) - use when the latency matters.

**The Garbage First GC** (G1) - new GC algorithm, for CMS replacement.

**What are types of reference in Java?**

Java provides 4 different types of Reference Objects: **strong**, **weak**, **soft** and **phantom**.

**Explain strong reference type in Java.**

**Strong reference**: This reference is something that we use daily while writing the code. Any object in the memory which has active strong reference is not eligible for garbage collection.

For example String s = "abc" , reference variable s has strong reference to String object "abc". Any object which has Strong reference attached to it is not eligible for garbage collection.

**Explain weak reference type in Java.**

Weak Reference are represented using java.lang.ref.WeakReference class and you can create Weak Reference by using following code.

WeakReference<StringBuilder> weakBuilder = **new** WeakReference<StringBuilder>(builder);

A weak reference is a reference that isn't strong enough to force an object to remain in memory. Weak references allow you to leverage the garbage collector's ability to determine reachability for you, so you don't have to do it yourself.

**Explain Serial Garbage Collector in Java.**

Serial garbage collector works by holding all the application threads. It is designed for the single-threaded environments. It uses just a single thread for garbage collection. The way it works by freezing all the application threads while doing garbage collection may not be suitable for a server environment. It is best suited for simple command-line programs.

Run with -XX:+UseSerialGC JVM argument to use the serial garbage collector.

Serial GC is recommended for client-style applications that do not have low pause time requirements.

**Explain Parallel Garbage Collector in Java.**

Parallel garbage collector also called as throughput collector, is the default garbage collector of the JVM. Unlike serial garbage collector, this uses multiple threads for garbage collection. Like serial garbage collector this also freezes all the application threads while performing garbage collection.

Turn on the -XX:+UseParallelGC JVM argument to use the CMS garbage collector.

The Parallel GC is recommended for situations where high throughput required.

**Explain CMS Garbage Collector in Java.**

Concurrent Mark Sweep (CMS) garbage collector uses multiple threads to scan the heap memory to mark instances for eviction and then sweep the marked instances. CMS garbage collector holds all the application threads in the following two scenarios only,

* While marking the referenced objects in the tenured generation space.
* If there is a change in heap memory in parallel while doing the garbage collection.

CMS collector uses more CPU to ensure better application throughput. If we can allocate more CPU for better performance, then CMS garbage collector is the preferred choice over the parallel collector.

Turn on the XX:+USeParNewGC JVM argument to use the CMS garbage collector

**Explain G1 Garbage Collector in Java.**

G1 garbage collector is used for large heap memory areas. It separates the heap memory into regions and does collection within them in parallel. G1 also does compacts the free heap space on the go just after reclaiming the memory. But CMS garbage collector compacts the memory on stop the world (STW) situations. G1 collector prioritizes the region based on most garbage first.

Turn on the -XX:+UseG1GC JVM argument to use the G1 garbage collector.

**What is soft reference in Java?**

A soft reference is exactly like a weak reference, except that it is less eager to throw away the object to which it refers. An object which is only weakly reachable (the strongest references to it are WeakReferences) will be discarded at the next garbage collection cycle, but an object which is softly reachable will generally stick around for a while.

**What is phantom reference in Java?**

A phantom reference is quite different than either SoftReference or WeakReference. Its grip on its object is so tenuous that you can't even retrieve the object -- its get() method always returns null. The only use for such a reference is keeping track of when it gets enqueued into a ReferenceQueue, as at that point you know the object to which it pointed is dead.

**Explain the javap command in Java.**

The javap command disassembles one or more class files. javap prints out the package, protected, and public fields and methods of the classes passed to it. javap prints its output to stdout.

You may use it to find out what methods are available for a class if you don?t have the source code that was used to create the class.

javap [options] JavaClassName

**What is PermGen in Java?**

PermGen stands for permanent generation space.

PermGen is a memory pool containing all the reflective data of the java virtual machine itself, such as class and method objects. With Java VMs that use class data sharing, this generation is divided into read-only and read-write areas. The Permanent generation contains metadata required by the JVM to describe the classes and methods used in the application.

The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.

**What is metaspace in JDK 1.8?**

PermGen has been completely replaced by MetaSpace in Java8, a native memory to store class meta-data information and that grows automatically.

We don't encounter **OutOfMemoryError** in Java 1.8, PermSize and MaxPermSize arguments are ignored at the start-up of application. On the contrary these two jvm arguments are now replaced by **MetaspaceSize**and **MaxMetaspaceSize.**These two new flags gives us the flexibility to change the value for default size of metaspace and to change the maximum value that this metaspace can take.

**Explain the usage of MetaspaceSize parameter in Java 8.**

Set MetaspaceSize to a value larger than the default, if you know that your applications needs more space for class data. Setting it to a larger size will avoid some number of GCs at startup.

**Difference between permgen and metaspace in Java.**

Metaspace auto increases its size by default (up to what the underlying OS provides), while PermGen always has a fixed maximum size.

**New features introduced in Java 1.7.**

1. Underscores Between Digits in Numeric Literals.

**int** ten\_million = **10**\_000\_000;

2. Improved Type Inference for Generic Instance Creation using **Diamond operator.**

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

// The above **statement** can be rewritten **using** Java Diamond **operator**.

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

3. Handling multiple exceptions in a single catch block.

4. String in Switch Expression.

5. Try-with-resources Statement.

6. Integral Types as Binary Literals.

**How do I load Jars dynamically at runtime?**

Using URLClassLoader we can load the jar.

URLClassLoader child = **new** URLClassLoader (myJarFile.toURL(), this.getClass().getClassLoader());

**Class** classToLoad = **Class**.forName ("net.javapedia.MyClass", **true**, child);

**Method** **method** = classToLoad.getDeclaredMethod ("methodToCall");

**Object** instance = classToLoad.newInstance ();

**Object** **result** = **method**.invoke (instance);

**What is synthetic construct in Java?**

Synthetic constructs are class, fields, and methods that are not defined in Java source code and created by the Java compiler while compiling.

For switch statement, Java compiler creates a synthetic field that start with $, and for the local classes which is required by the VM.

Synthetic constructs are also known as dynamic proxy.

**Difference between Java.exe and Javaw.exe.**

Java is the plain old java interpreter, you have a console window open all the time. Javaw is a wrapper around java that doesn't open a console window.

java.exe pops up a console window. javaw.exe does not. If you expect text output, you'll need to use java.exe. If the application pops up its own window (For example, Eclipse), you can use javaw.exe.

**What is Nashorn in Java8?**

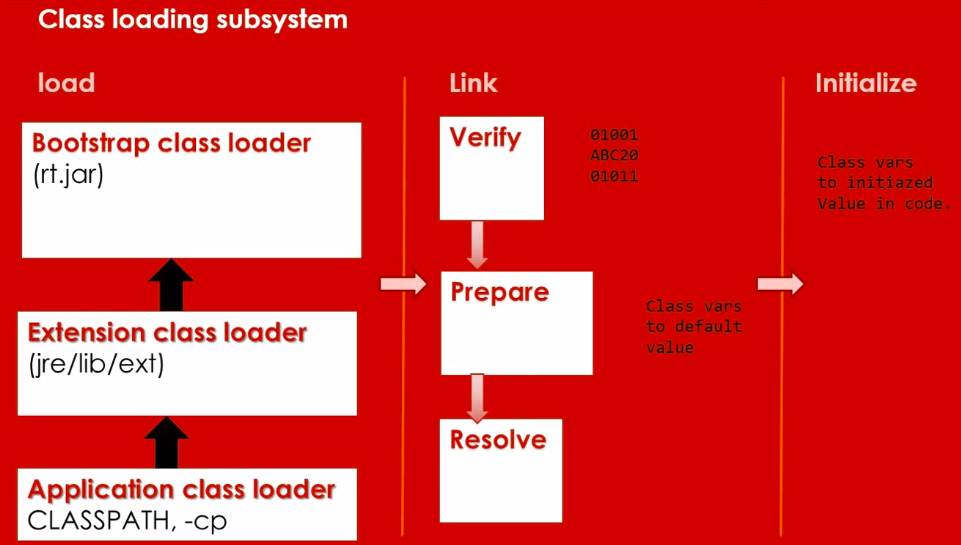
Nashorn is the new Javascript processing engine for the Java platform that shipped with Java 8. Until JDK 7, the Java platform used Mozilla Rhino for the same purpose as a Javascript processing engine.

**How do I debug to see when class is loaded in jvm?**

You may add the command line option -verbose:class to your Java process, this will display information about each class loaded.

java -**verbose:class**

**Explain Class loading subsystem of JVM.**



**Can a Java source file have more than one class declaration?**

Yes. However only one class can be declared as public and the java file name should be exactly same as the public class name along with .java extension.

**How to implement cloning?**

The java.lang.Cloneable interface must be implemented by the class whose objects would be cloned.

**Implement the clone () method in the** class protected Object clone () throws CloneNotSupportedException.

**Object Cloning.**

Object cloning is one of the ways to create an exact copy of an object.

Example of a final class in Java API.

java.lang.System

**What is the data type of System.out?**

out is of type Print Stream and it is a static member in the System class.

**What is the base class of all classes?**

java.lang.Object class.

**Could a Java class file exist even without filename and only extension(.java)?**

Yes. The file is to be saved as .java without any file name.

compile using **javac .java** and run using **java <className>**

For e.g., .java file content:

**class** **JavaClass** {

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

System.out.println(".java file content");

}

}

**Difference Between ClassNotFoundException & NoClassDefFoundError.**

Both errors are related to missing classes in the class path, however the main difference lies how it originate.

ClassNotFoundException occurs when you try to load a class at runtime by using Class.forName() or loadClass() and requested class is not present in class path. It is thrown when an application tries to load in a class through its name, but no definition for the class with the specified name could be found.

NoClassDefFoundError is encountered when the class was available at compile time but not during runtime.

ClassNotFoundException Example.

**package** com.tutorials.classes;

**public** **class** **ExampleClassNotFoundException** {

**private** **static** **final** String CLASS\_TO\_LOAD = "com.tutorials.classes.NotCreatedClass";

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

**try** {

Class loadedClass = Class.forName(CLASS\_TO\_LOAD);

System.out.println("Class " + loadedClass + " found successfully!");

} **catch** (ClassNotFoundException ex) {

System.err.println("ClassNotFoundException was caught: "+ ex.getMessage());

ex.printStackTrace();

}

}

}

|  |  |
| --- | --- |
| **ClassNotFoundException** | **NoClassDefFoundError** |
| It is an exception. It is of type java.lang.Exception. | It is an error. It is of type java.lang.Error. |
| It occurs when an application tries to load a class at run time which is not updated in the classpath. | It occurs when java runtime system doesn’t find a class definition, which is present at compile time, but missing at run time. |
| It is thrown by the application itself. It is thrown by the methods like Class.forName(), loadClass() and findSystemClass(). | It is thrown by the Java Runtime System. |
| It occurs when classpath is not updated with required JAR files. | It occurs when required class definition is missing at runtime. |

**Dynamic class loading.**

A class can be loaded using one of the following methods:

* Class.forname,
* ClassLoader.findSystemClass,
* ClassLoader.loadClass.

**Difference: Class.forName() vs ClassLoader.loadClass().**

Class.forName() uses the caller's classloader and initializes the class (runs static intitializers, etc.).

loadClass is a ClassLoader method, so it uses an explicitly-provided loader, and initializes the class lazily (on first use).

**Which class implements clone method- Cloneable or Object in Java?**

Java.lang.Object implements the clone() method.

Clonable is a marker interface and it does not have any methods.

Object.clone() is a native method implemented using C, C++ or any other native language.

**How does Java ClassLoader Work?**

When JVM requests a Java class, it invokes loadClass method of the ClassLoader by passing the fully classified name of the Class. loadClass method calls findLoadedClass() method to check that the class has been already loaded or not.

If the Class is not already loaded then it will delegate the request to parent ClassLoader to load the class.

If the parent ClassLoader does not find the Class then it will invoke findClass() method to look for the class in the filesystem.

**When a class is loaded in Java?**

Class loading is performed by ClassLoaders in Java which can be implemented to eagerly load a class as soon as another class references it or lazily load the class until a need of class initialization occurs.

If a Java class is loaded before its being used it resided at JVM before being initialized. This differs between JVM to JVM. While its guaranteed by JLS that a class will be loaded when there is a need of static initialization.

**When a class is initialized in Java?**

After class loading, initialization of class takes place by initializing all static members of class.

A Java class's static initialization usually happens immediately before the first time one of the following events occurs:

* an instance of the class is created,
* a static method of the class is invoked,
* a static field of the class is assigned,
* a non-constant static field is used, or
* for a top-level class, an assert statement lexically nested within the class is executed.

Reflection can also cause initialization of class. Some methods of java.lang.reflect package may cause class to be initialized.

**When a class is initialized in Java?**

After class loading, initialization of class takes place by initializing all static members of class.

A Java class's static initialization usually happens immediately before the first time one of the following events occurs:

* an instance of the class is created,
* a static method of the class is invoked,
* a static field of the class is assigned,
* a non-constant static field is used, or
* for a top-level class, an assert statement lexically nested within the class is executed.

Reflection can also cause initialization of class. Some methods of java.lang.reflect package may cause class to be initialized.

**What are the rules of class initialization in Java?**

Classes are **initialized from top to bottom fashion**so that fields declared on top initialized before field declared in bottom.

Super Class is initialized before Sub Class or derived class is initialized in Java.

If Class initialization is triggered due to the access of a static field, **only Class which has declared static field is initialized and it doesn't trigger initialization of super class or sub class** even if static field is referenced by type of Sub Class, Sub Interface or by implementation class of interface.

interface initialization in Java doesn't cause super interfaces to be initialized.

static fields are initialized during static initialization of class while non-static fields are initialized when instance of class is created. It means static fields are initialized before non-static fields in Java.

non-static fields are initialized by constructors in Java. sub class constructor implicitly calls super class constructor before doing any initialization, which guarantees that non-static or instance variables of super class is initialized before sub class.

**What is a Local class in Java?**

The class defined inside a block is called local class. Such a class has local scope and isn't usable outside the block where it is defined.

**Different type of cloning in Java.**

Java supports 2 types of cloning: - **Deep** and **shallow** cloning. By default, shallow clone is used in Java. Object class has a method clone () which does shallow cloning.

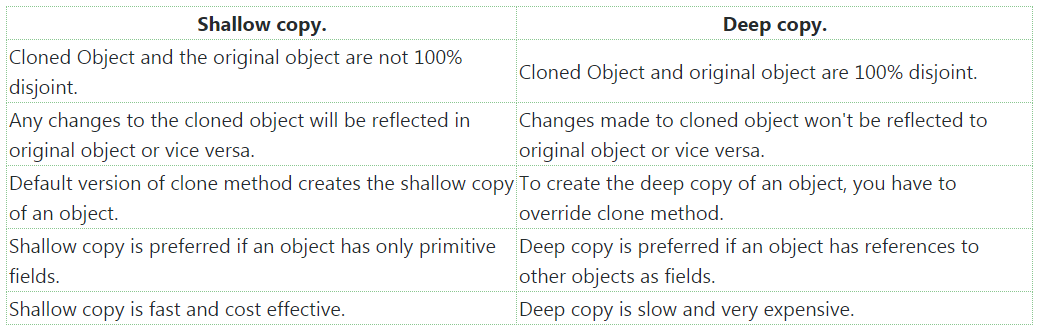
**Explain Shallow copy in Java.**

Shallow clone is a copying the reference pointer to the object, which mean the new object is pointing to the same memory reference of the old object. The memory usage is less in case of shallow copy.

**Explain deep copy in Java.**

Deep copy of an object will have exact copy of all the fields of original object just like shallow copy. But in additional, if original object has any references to other objects as fields, then copy of those objects are also created by calling clone () method on them. That means clone object and original object will be 100% disjoint. They will be 100% independent of each other. Any changes made to clone object will not be reflected in original object or vice-versa.

To create a deep copy of an object, you must override the clone () method.



**What is Serialization in Java?**

Serialization is a process of writing an Java Object into file along with its attributes and content. It internally converts the object in stream of bytes

**Explain De-serialization in Java.**

De-Serialization is a process of reading the Java Object and its properties from a file along with the Object's content.

**Difference between readObject and readResolve in Java serialization.**

The readObject method is responsible for reading from the stream and restoring the classes fields.

readResolve is used for replacing the object read from the stream. This helps enforcing singletons; when an object is read, replace it with the singleton instance. This ensures that nobody can create another instance by serializing and deserializing the singleton.

The readResolve method is called when ObjectInputStream has read an object from the stream and is preparing to return it to the caller. ObjectInputStream checks whether the class of the object defines the readResolve method. If the method is defined, the readResolve method is called to allow the object in the stream to designate the object to be returned.

**Difference between Static block and initializer block in Java.**

The static initializer block will be called on loading of the class and will have no access to instance variables or methods. It is often used to create static variables.

The non-static initializer block is created on object construction only, will have access to instance variables and methods. It will be called at the beginning of the constructor, after the super constructor has been called (either explicitly or implicitly) and before any other subsequent constructor code is called.

What is static class in Java?

Static class is a Java class, which only contains static methods. An example of static class is java.lang.Math,which contains utility static methods for various math features such as sqrt, random.

**Role of Runtime and System class in Java.**

**Runtime** class provides access to the Java runtime system to retrieve information like memory availability, invoking the garbage collector, etc.

**System** class provides access to system resources. It enables access to standard input/output, error output streams, current time in millis, terminating the application, etc.

**What are the Object class methods?**

protected Object clone () throws CloneNotSupportedException

Creates and returns a copy of this object.

public boolean equals (Object obj)

Indicates whether some other object is "equal to" this one.

protected void finalize () throws Throwable

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

public final Class getClass ()

Returns the runtime class of an object.

public int hashCode()

Returns a hash code value for the object.

public String toString()

Returns a string representation of the object.

The notify, notifyAll, and wait methods of Object all play a part in synchronizing the activities of independently running threads in a program. There are 5 of these methods:

public final void notify() public final void notifyAll () public final void wait() public final void wait(long timeout) public final void wait(long timeout, int nanos)

**How to implement equals method correctly?**

Consider that you are overriding equals method for Employee Class having empId, firstName and lastName.

1. Check for reference equality, if equal then return true.
2. Check for null check on the object argument being passed, if null return false.
3. Compare getClass methods, if different return false.
4. Always compare numeric and identify fields first.
5. Check for field's references first and then check for null and equality.
6. Override hashcode method whenever you override equals method.

The above steps are illustrated in the below example.

**@Override**

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

**if** (**this** == o) // check 1

**return** **true**;

// null check

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

**return** **false**;

// type check and cast

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

**return** **false**;

Employee emp = (Employee) o;

//numeric and identity fields first

**if** (!Objects.equals(**this**.emplId, emp.emplId)) **return** **false**;

// field comparison

**return** Objects.equals(firstName, emp.firstName)

&& Objects.equals(lastName, emp.lastName);

}

**Can two objects which are not equal have the same hashCode?**

Yes, possible. Two objects which are not equal to equals () method can still return same hashCode.

**What happens when equals () is not consistent with compareTo() method?**

java.util.Set implementations such as SortedSet, TreeSet uses compareTo method for comparing objects. When compareTo does not return 0 while equals are true, it breaks Set contract and may result in **duplicates**.

**Difference between instance of and getClass () method for checking type inside equals.**

**instance of** operator returns true, even if compared with subclass, for example, Subclass instance of Superclass is true, but with **getClass**() its false. By using getClass () you ensure that your equals () implementation doesn't return true if compared with subclass object.

**Is it possible to access a non-static variable in static context?**

A static variable in Java is at class level and shared among all its instances. A static variable is initialized when the class is loaded by the JVM.

When we try to access a non-static (instance-level) variable from the static context, we encounter compilation error as instance variables are not created and it is not associated with any instance.

**What is transient variable in Java?**

Transient variable cannot be serialized.

For example, if a variable is declared as transient in a class implementing Serializable interface and the class is persisted to an Object Stream, the value of the variable is ignored and not written to the stream. When the class is retrieved from the Object Stream the value of the variable becomes null.

**What is shadowing in Java?**

Shadowing refers to the practice of using two variables with the same name within scopes that overlap. When shadowed, the variable with the higher-level scope is hidden because the variable with lower-level scope overrides it. The higher-level variable is then shadowed.

You can access a shadowed class or instance variable by using its fully qualified name which is the name of the class that contains it. Shadowing is also known as variable shadowing.

**public** **class** **Shadowing** {

**int** i = **5**;

**public** **void** **setI** (**int** i) {

**this**.i = i;

}

}

In the above example, the variable 'i' is shadowed in the setter and this keyword is used to refer to the instance variable 'i'.

**Different types of variables in Java.**

* Instance or non-static variables,
* Static or class variables,
* Local or method variables,
* and parameters.

**Difference between static type and dynamic type programming languages.**

A language is statically typed if the type of a variable is known at compile time. For example, in Java, C and C++, programmer must specify what type each variable is.

A language is dynamically typed if the type is associated with run-time values. Example programming languages are perl, python and ruby.

**What are local variables in Java?**

A local variable is one that exists inside a method or a block of code and exists if that method or block is executing. Once the program reaches the end of the method (or block), the local variable disappears from memory. The next time the method is called, a completely new version of the local variable comes into existence. One of the most common types of local variables is a parameter.

**Is local variable thread safe in java?**

Yes. All local variables defined in your program will be allocated memory in the stack. So, when you create a thread it will have its own stack created. Two threads will have two stacks and one thread never shares its stack (that is, local variables) with other thread.

**How do I define a constant?**

The variable should be declared with static and final modifiers. static ensures that only one variable exists for all instances of the class and final makes its value not changeable once initialized.

**static** **final** **int** MAX\_HEIGHT = **50**;

**What are static variables?**

Static variables are class level variable where all instances of the class refer same variable. When an instance updates its value, all the other objects will see the new value.

**Explain static keyword in Java.**

The static keyword can be applied to,

* static variables,
* static methods,
* static block,
* static inner class,
* and interface static method (introduced in Java 8).

**What is a static class in Java?**

A Class can be made static only if it is a nested class (class within a class). The nested static class can be accessed without having an object of outer class.

It can access static data members of outer class including private. Static nested class cannot access non-static data member or method.

If you have the static member inside static nested class, it can be accessed directly, and you don't have to create instance of static nested class.

The static class are also known as static nested class or static inner class.

**class** **OuterClassEx**{

**static** **int** val=**30**;

**static** **class** **InnerClassEx**{

**void** **msg**(){System.out.println("Value is "+val);}

}

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

OuterClassEx.InnerClassEx obj=**new** OuterClassEx.InnerClassEx();

obj.msg();

}

}

**What is a static block?**

A static block, is a block of code inside a Java class that will be executed when a class is first loaded into the JVM. Mostly the static block will be used for initializing the variables.

Static block will be called only one while loading and it cannot have any return type also not have 'this' or 'super' keyword.

**public** **class** **TestStaticBlock** {

**static** **int** val;

**static** {

val = **100**;

System.out.println("Hello from static block. val ="+ val);

}

}

**Can we serialize static variables?**

No, Static variables are shared variables and don't correspond to a specific object.

**What is a static import?**

Static imports allow us to import all static fields and methods of the class, you can access them without class name reference. static import is introduced in Java 5.

import static java.lang.System.out;

import static java.lang.Math.\*;

class WithStaticImports {

public static void main(String [] args) {

out.println("round " + round(1032.897));

out.println("min " + min(60,102));

}

}

**Can we execute a program without main () method?**

Yes, it is possible in previous versions of Java, not since JDK 1.7 using static block.

**How the object references are initialized when it is an instance variable?**

Object references are initialized to null in case of instance variables.

**Define local variable.**

Local variable also known as block level variable, are identifiers that are declared within a block of code or method.

This variable need to be initialized before using it.

In the below examples, "i" variable is an example of local variable and it is visible only to the enclosing block.

{

**int** i;

}

**void** **methodName**() {

**int** i;

}

**Difference between final and effectively final in Java.**

The keyword final ensures that the variable is **initialized only once and cannot be altered**. Reinitializing would issue compilation errors.

Any variable that is **initialized only once and not marked as final**is known as effectively final. This variable when marked final will not issue any compilation errors.

**What is the difference between the Boolean & operator and the && operator?**

|  |  |
| --- | --- |
| & | && |
| & is the "bit-wise AND" operator. | && is the "conditional logical AND" operator. |
|  |  |
| & always evaluates both arguments. | && evaluates the first argument. if it is true, it then evaluates the second. |

**Explain Diamond Operator in Java.**

<> denotes the diamond operator. Purpose of the diamond operator is to simplify instantiation of generic classes.

For example, see the below.

List<Integer> myList = **new** ArrayList<Integer>();

//with the diamond operator we can write as below easily.

List<Integer> myList = **new** ArrayList<>();

The Diamond Operator reduces some of Java's verbosity surrounding generics by having the compiler infer parameter types for constructors of generic classes.

**Write a Java program to generate a random number between 0 to 99.**

The below program uses the current time in milliseconds and apply modulo operator to pick a random number between 0 to 99.

public class RandomNumberGen {

public static void main(String[] args) {

long start\_time = System.currentTimeMillis();

System.out.println(start\_time % 100);

}

}

**What is the difference between ++i and i++ under increment operators?**

In ++i, the value of i will get incremented before it is used in the expression.

In i++, the previous value is used in the expression first, and after that i is modified and incremented.

**Difference between Java IO and NIO packages.**

|  |  |
| --- | --- |
| **IO.** | **NIO.** |
| Stream oriented. | Buffer oriented. |
| Blocking IO. | Non-blocking IO. |
| No Selectors. | Selectors available. |

**Why does the main () method declared static?**

JVM need to access the main () method even before the class instantiation so begin the execution. So, it is declared static.

**Explain the Default method for interfaces in Java 8.**

In Java 8, a new feature "default method implementation for interfaces" is introduced that **facilitates backward compatibility** for the old interfaces to leverage the lambda expression capability of Java 8 and existing libraries implementing these interfaces **need not have to provide its implementation for the new functionality/method added to the interface.**

For example, java.util.List or Collection interface does not have forEach method declaration. Thus, calling such methods will break the collection framework implementations. Java 8 introduces default method so that List/Collection interface can have a default implementation of forEach method, and the class implementing these interfaces need not have to implement the same.

Thus, default methods enable you to add new functionality to the interfaces of your libraries and ensure backward compatibility with the codes written using the older versions of those interfaces without having to implement the new functionality.

For creating a default method in Java interface, we need to use **default keyword**with the method signature.

**public** **interface** **Fruits** {

**default** **void** **nature**(){

System.out.println("I am Sweet.");

}

}

The **default methods are non-static** and can be accessed from the Implementing class object as shown in the below example.

**public** **class** **OrangeClass** **implements** Fruits {

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

Fruits fruit = **new** OrangeClass();

fruit.nature();

}}

Java interface default methods are also known as **Defender Methods or Virtual extension methods**.

A default method **cannot override a method from java.lang.Object class** as interfaces does not extend Object.

Compilation error occurs when a class tries to implement two or more interfaces having a default method with the same signature.

Other than multiple inheritance, there is no significant difference between an abstract class and an interface in Java 8 by the addition of default method feature.

**Advantages of default method in Interface feature in Java 8.**

* Using default methods for interfaces **eliminates the need for utility classes**, for example, the java.util.Collections utility class is not required when all of its methods are provided in the Collection interface itself as default methods.
* It helps in **extending interfaces** without the concern of breaking the implementation classes.
* Default methods in interfaces enhances the Collections API in Java 8 to **support lambda expressions**.
* Java interface default methods has **bridged down** the differences between interface and abstract class.
* Interface implementing concrete classes can choose **which default method to override** and can use interface default implementation itself.

Explain the static interface methods feature in Java 8.

The interface static methods are **concrete methods** implemented in the interface that **prevents the implementation classes from overriding to ensure the proper and uniform implementation** being used across all the implementation classes.

The static interface method has method body and marked with **static keyword** in the method signature.

**public** **interface** **Fruits** {

**static** **void** **eat**() {

System.out.println("Enjoy!");

}

}

Java interface static methods are**visible to interface methods only** and it can be **invoked by interface name**itself.

**public** **class** **OrangeClass** **implements** Fruits {

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

Fruits.eat();

}

}

**Advantages of using static interface method in Java 8**.

* Java interface static methods are suitable for providing utility function and acts as utility methods, for example sorting a Collection, validations, reversing a collection etc.
* Java interface static method ensures quality and security by preventing the implementation classes to override them.

**Difference between default and static methods in Java interface.**

|  |  |
| --- | --- |
| **Default method.** | **Static method.** |
| default method are used as a default implementation for classes that implements that interface. | static method in interface is used as Helper methods. |
|  |  |
| default methods can be invoked by the implementation class objects. | static methods are like static class methods and can be invoked using Interface name. |

**Difference between System.exit (0), exit (1) and exit (-1) in Java.**

Zero represents that the program ended successfully. Any number greater than zero represents program execution failed. Number less than zero represents program execution error.

**What is virtual function/method in Java?**

In Java, all non-static methods are considered as **virtual functions**. The Final methods cannot be overridden, and the private methods cannot be inherited so it is considered as non-virtual.

**What is lambda expression in Java 8?**

A lambda expression is an **anonymous function**that you can use to create delegates or expression tree types. By using lambda expressions, you can write local functions that can be passed as arguments or returned as the value of function calls. A lambda expression is the most convenient way to create that delegate.

Explain the scenario how Default Method can cause Multiple Inheritance Ambiguity Problem.

Java class can implement multiple interfaces and each interface can define default method with same method signature, therefore, the inherited methods can conflict with each other.

**public** **interface** **Interface1** {

**default** **void** **myDefaultMethod**(){

System.out.println("Interface One default method");

}

}

**public** **interface** **Interface2** {

**default** **void** **myDefaultMethod**(){

System.out.println("Interface Two default method");

}

}

**public** **class** **ImplClass** **implements** Interface1, Interface2 {

}

The above code will fail to compile with the error. To fix this class, we need to provide default method implementation.

**public** **class** **ImplClass** **implements** Interface1, Interface2 {

**public** **void** **myDefaultMethod**() {

}

}

Additionally, we may invoke default implementation provided by any of super interface by using super keyword as shown below.

**public** **class** **ImplClass** **implements** Interface1, Interface2 {

**public** **void** **myDefaultMethod**() {

Interface2.super.myDefaultMethod();

}

}

**Why Java doesn't allow overriding of static methods?**

Overriding depends on having an instance of a class and static method is not associated to instance of the class.

**Explain public static void main (String args[]) method signature.**

**public** access modifier specifies who can access this method. Public will be accessible by any Class from any package.

**static**keyword identifies it is class based and it can be invoked without creating a class instance.

**void** is the return type of the method. Void indicates that the method does not return any value.

**main** is the name of the method which is invoked by JVM as a starting point of execution for an application.

**String args[]**holds the command line parameters passed to the main method.

**What are the primitive data types in Java?**

There are eight primitive data types.

* byte.
* short.
* int.
* long.
* float.
* double.
* boolean.
* char.

**What is Autoboxing and Unboxing?**

Autoboxing refers to the automatic conversion of primitive types to its corresponding object wrapper classes.

For example, converting an int to an Integer, a double to a Double etc.

This automatic conversion is accomplished by the compiler.

Character ch = 'c';// char autoboxed to its wrapper class.

Float fl = **2.5f**; //float converted to Float object

**Is Java primitive data type stored on stack or heap?**

Primitive types declared locally will be on the stack while primitive types that are defined as part of an object instance are stored on the heap.

Local variables are stored on stack while instance and static variables are stored on the heap.

Difference between double and float variables in Java.

In java, float takes 4 bytes in memory while Double takes 8 bytes in memory. Float is single precision floating point decimal number while Double is double precision decimal number.

What is the default value of char data type in Java?

The default value of a char primitive type is '\u0000'(null character) as stated in the Java Language Specification.

The shortcut for 'u0000' is '\0', So the null can be represented either by 'u0000' or '\0'.

The below Java program validates null representations using instance char field 'c'.

public class DefaultValueForchar {

char c;

public static void main(String[] args) {

char c0 = '\0';

char cu0000 = '\u0000';

DefaultValueForchar obj = new DefaultValueForchar();

System.out.println(obj.c);

System.out.println(c0);

System.out.println(cu0000);

System.out.println(c0==cu0000);

System.out.println(obj.c==c0);

System.out.println(obj.c==cu0000);

}

}

When does autoboxing and unboxing occur in Java?

Autoboxing and unboxing can happen where an object is expected and primitive type is provided, for example, In a method invocation where an object argument is expected but primitive values are provided, Java automaticallyconverts primitive into its equal value Object.

lassic use of autoboxing is adding primitive types into Collection like ArrayList in Java.

ArrayList<Integer> intsList = **new** ArrayList<Integer>();

intsList.add(**1**); //autoboxing - primitive to object

intsList.add(**2**);

When does autoboxing and unboxing occur in Java?

Autoboxing and unboxing can happen where an object is expected, and primitive type is provided, for example, In a method invocation where an object argument is expected but primitive values are provided, Java automaticallyconverts primitive into its equal value Object.

lassic use of autoboxing is adding primitive types into Collection like ArrayList in Java.

ArrayList<Integer> intsList = **new** ArrayList<Integer>();

intsList.add(**1**); //autoboxing - primitive to object

intsList.add(**2**);

Top of Form