1. How do you define Object-Oriented Programming?

It's a style of programming; some call it a programming paradigm; which sends messages between objects trying to simulate the evolution of real-world objects. It is a viewpoint where it looks at real-world space as a set of objects, some simple with primitive functionalities and attributes, and some complex with more complicated behaviours. It tries to create a similar structure in the code base, where these objects can be related to each other, can follow similar rules, and can evolve.

2. What are different language type systems? What is Java?

- Statically typed: data types checked at compile time
- Dynamically typed: checked at runtime

- Manifestly typed: variables must be declared
- Type inferred: variable types are deducted by context

- Strongly typed: types are restricted within their own bound. Not convertible.
- Weakly typed: a value of one type can be treated as another

Java is a strongly, manifestly statically typed language.

3. What are Compiled and Interpreted language? What are Java and JVM?

In compiled languages (like C and C++), the source code we write will be converted to binary machine code by a compiler and then gets executed and so it is platform dependent. On the other hand, in an Interpreted language like Bash or SQL, the script is platform-independent and can get executed only if the interpreter is available. Java is both compiled and interpreted language since our source code will be converted to Java Byte Code, not machine code, and sent to the client (which could be any operating system) and then will be interpreted by Java Virtual Machine (JVM).

4. What is an IDE?

IDE, which stands for Integrated Development Environment, is an automation or suggestion tool that performs many normal tasks automatically like compilation, the project setup, and early warnings. That's a tool to aid productivity so it doesn't mean that we cannot work without it, but it helps a lot.

5. What is the difference between a class and an object?

A class is a blueprint or template for creating objects. You can have just one class which defines how your objects would be, and then create multiple objects from your class using the new keyword. This process of creating objects from a class is called instantiation and each of those objects is one instance of the class.

6. What are value and reference types? How does Java pass variables?

All the primitive type variables are value type, and anything other than of those is reference-type. In Java all object variables, including String, is reference type which means the object variable does not contain object data, rather it holds only the reference to the location of the actual object.

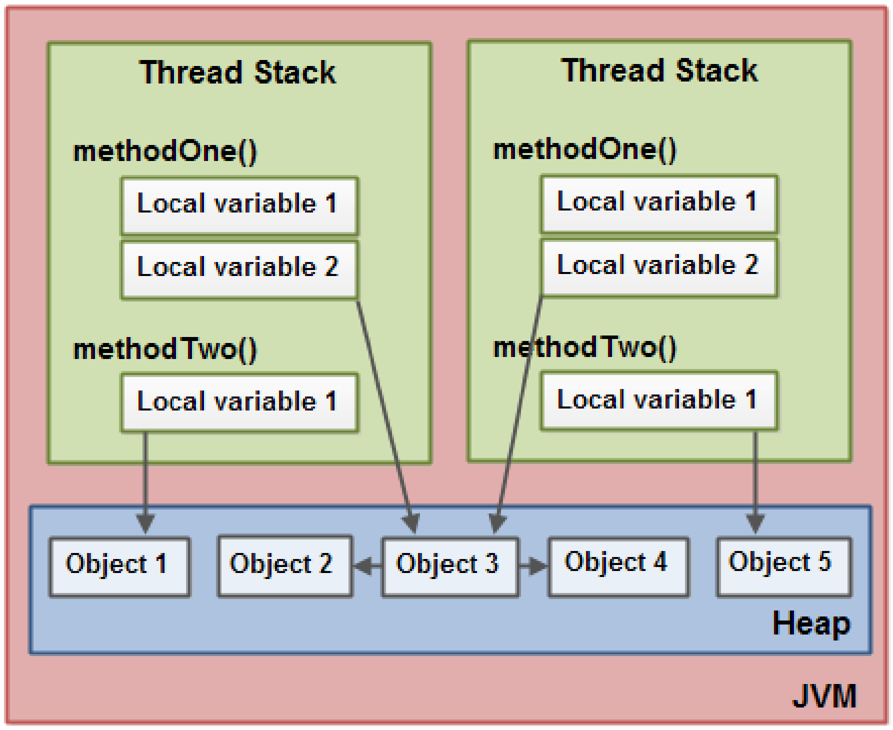
Java is strictly only a type of pass by value. It means that when passing a primitive type to a function, a copy of the value of that variable will be passed and the value of the variable itself will not change, no matter what changes made inside the function since that is another copied variable. Also, when passing a reference type variable, it again passes the value, but this value, itself, is a reference or address to a memory location which holds the actual data. So if we do some changes to the actual data using the passing reference, it will affect the variable's value.

7. What is the difference between stack and heap memory?

The stack is used for storing primitive type variables, method calls, and the reference for objects. The stack follows the LIFO order (Last In First Out) so memory will be immediately reclaimed after it goes out of scope. There is one stack per thread and if due to an infinitive recursive method calls, it gets full, we call it stack overflow.

But for reference type variables which are more complex, the object, itself, is saved in the heap. This variable on the stack just saves the address of the object in the heap memory, not any data or value of it. Objects stored in the heap get removed later on when they're no longer any references to them. This is done by Java's garbage collector. There is only one heap per JVM and the string pool is also a part of the heap.

The Java Garbage Collector runs in its own thread and stops the application completely while running. That's the main reason why standard JVM is not a good option in real-time applications.



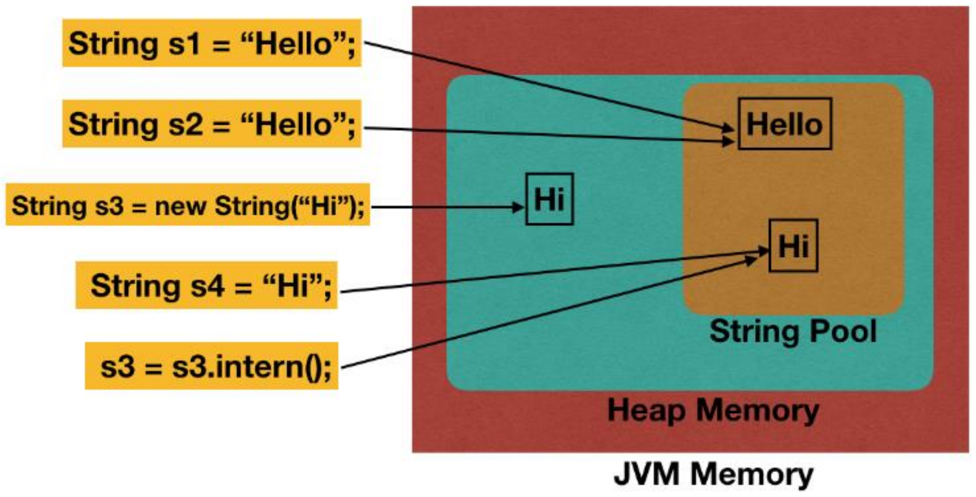
8. What do we know about Strings in Java?

They are reference-type, not primitive, so the value of a string variable itself is an address or reference to the memory location where the actual set of characters as a string are stored. Therefore, we cannot use == to check 2 strings' equality, instead, the .equals method which compares two objects' equality must be used.

Moreover, strings are immutable, which means they never change. Any function that does any kind of manipulation on a string, returns a new string as the result. In case of not facing this issue, StringBuffer (Thread-safe but slower) and StringBuilder (Not thread-safe but faster) can be used which both allocate an initial capacity for the string but it is resizable.

Besides, there is a String pool in Java which holds all the strings created in a specified location of memory. Anytime we use the new keyword to initialize a string, a new string object, with a new reference, will be created in the string pool.

But if we are just assigning a set of characters to a string to initialize it, the new object will be created only if there isn't any string with that value in the pool, otherwise, a reference to the existing string object will be assigned to that variable.



9. How does OOP help solve the procedural code's problems?

In procedural code, we have big classes with several unrelated methods focusing on different concerns and responsibilities. Also, these methods often have several parameters and you might see the same group of parameters passing to different methods repeatedly.

By applying object-oriented programming techniques, we remove these repetitive parameters passing, instead, by classifying and declaring them in a class we will group related fields and functions together. Our classes are encapsulating them and then by implementing the abstraction concept, we'll make sure to show as minimum as possible detail of what is happening inside of our class to the user and other classes. It reduces complexity and makes the code more reusable. Then inheritance and polymorphism help us to categorize our classes, remove repeated lines and big switch-case statements. Somehow we are simulating what's in the real world and nature, so the codding would be faster and the result would be cleaner and more understandable.

10. What is Encapsulation?

The encapsulation, also called data hiding, is that we should group the data and operations on the data inside a single logical unit (class) together. The more bundling related components together, the more cohesion, the less coupling, the more preferable. Encapsulation ensures that implementation details are not visible to users and the variables of one class will be hidden from the other classes, accessible only through the methods of the current class.

11. Why should we declare fields as private?

For variables, we don't want to be changeable from other parts of the code or variables should be changed only by special functions, like getters and setters, we need to declare them as private. It will help us to control the process. For example, in a banking system, the balance of an account must be accessible only for functions of deposit and withdrawal. Nothing else should be able to change it so it should be private.

12. What is Coupling and what are its effects?

Coupling represents the level of dependency between software entities (like classes). The more our classes are dependent on each other or coupled together, the harder it is to change them. The optimal situation is to reduce coupling as long as it doesn't make any problem for the logic or algorithm.

13. What is Abstraction?

Abstraction suggests that we should reduce complexity by hiding the unnecessary implementation details. When a driver starts a car, he doesn't wanna know what is exactly happening inside the car and engine. It's better to keep the interface as simple as possible.

Moreover, the abstraction says that each significant functionality of a program should be implemented in just one place in the source code, so if we are repeating something, it should be abstracted out before the third and fourth copies come.

14. How does the abstraction principle help reduce coupling?

When we just show the abstract interface of a class to others, so they are not coupled to its implementation details. In this way, we prevent other classes from getting affected when we change these details. For example, if we change something inside a remote control to a new model, we're not affected. We still use the same interface to work with our TV. Also, reducing these details and exposing fewer methods makes our classes easier to use. For example, remote controls with fewer buttons are easier to use.

15. What is Inheritance?

Inheritance is the act of receiving the behaviors and attributes of another class or interface. This would be an IS-A relationship which is performed by Realization (inheriting from something abstract), and Generalization (inheriting from a concrete entity).

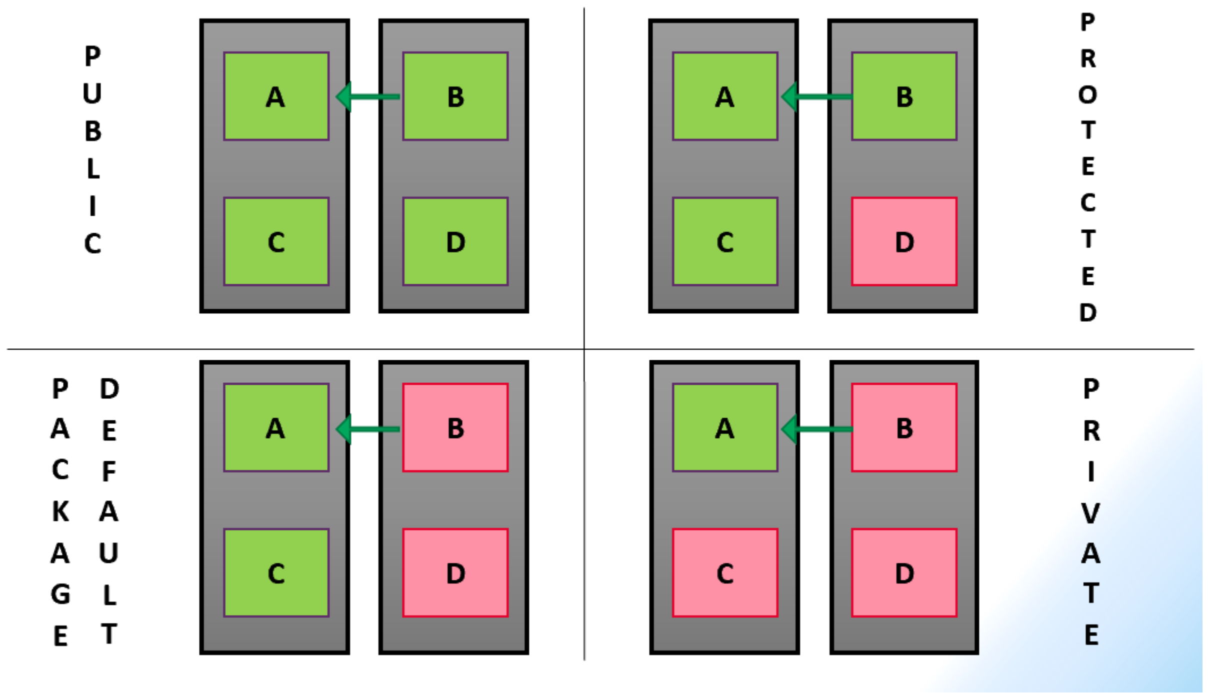
16. What is polymorphism?

Polymorphism, meaning having different forms, is related to use the same function name for different situations. Totally, there are 2 types:

- ✓ method Overriding, also called runtime polymorphism or late binding, in which you have the same signature but in different classes in a hierarchical order resulting in different implementations for each call based on the object that the method is calling for. So the implementation of the method call might change in the runtime, that's why we call it late binding.
- ✓ The other type is method Overloading, also known as compile-time polymorphism or early binding, which happens in a single class for functions with the same name but different signatures. The overloaded functions can take different combinations of parameters.

17. What is the difference between access modifiers?

- ✓ For **classes**, the available modifiers are public or default (left blank), as described below:
  - **public**: The class is accessible by any other class.
  - **default**: The class is accessible only by classes in the same package.
- ✓ The following choices are available for **attributes and methods**:
  - **default**: (left blank) is available to any other class in the same package.
  - **public**: Accessible from any other class.
  - **protected**: same as the default access modifier, with the addition that subclasses can access protected methods and variables of the superclass
  - **private**: Accessible only within the declared class itself.



18. What do the Final and Static keyword do?

The final keyword before a class makes it unable to be extended. Before a method, we cannot override it; and before a variable makes it constant after initialization.

The static keyword makes a function or variable a member of the class, itself, not a member of objects or instances. So, it's the same for all objects and we don't need to have any object to be able to call the method or use the variable, we can simply call them via the class name.

19. What are Constructors?

Constructors are the first functions called when we create a new object. We use them to prepare the object to use and initialize our variables. They have no return types, same name, and same access level of the class. They can be overloaded as needed but all classes must have at least 1 constructor. If we have defined our constructor, it will be called when instantiating, otherwise, java creates a simple empty constructor with no passing arguments and calls it, itself. This is called the default constructor.

Moreover, all constructors must start with either super(), which calls the parent class constructor, or this(), which calls another constructor of the same class. Only 1 constructor call is permitted inside each constructor so sometimes we will have a constructors-chaining.

20. What is the Object class? What are good to override from that?

The object class is the super parent of all other classes in Java, either directly or indirectly. There are some useful functions inherited from the Object class and is a good practice to override them as we want:

- HashCode(): returns a numeric value that is by default calculated based on the address of the object in memory so it would be different for each object. When overriding, we can calculate the hashCode based on the object's attributes so it would be helpful when comparing objects from this class.
- Equals(): Whenever we want to be able to compare 2 objects, we can override these 2 functions. In the equals() function we can define the exact way of comparison based on our desired attributes.

- ToString(): This is the method that gets called automatically whenever we pass on an object for being printed. By default, this returns something based on the result of the hashCode() function, so it's better to re-implement it (override) in the order we want.

21. What does the Abstract keyword do?

The Abstract keyword before a class declaration makes it unable to be instantiated, and instead, it needs to be extended to share the common code. An abstract class might have abstract methods, which do not have any implementation, it's just the signature, and then the abstract methods must be overridden in the first concrete subclasses.

An abstract method must be in an abstract class but an abstract class might have no abstract methods. They are usually used when sharing a common basic functionality with subclasses when interfaces can be implemented by any of them as some upgrades or additional features.

22. What is an Interface? Why do we use them?

An interface is a set of methods which are implicitly public and abstract. Any abstract method of the interface is guaranteed to be provided a body function by the classes implementing this interface. So we are making our class follow this abstract contract and describing how the outside world can interact with the class. Not only does this organize our classes, but also coding against interfaces makes our application loosely-coupled, extensible, and testable.

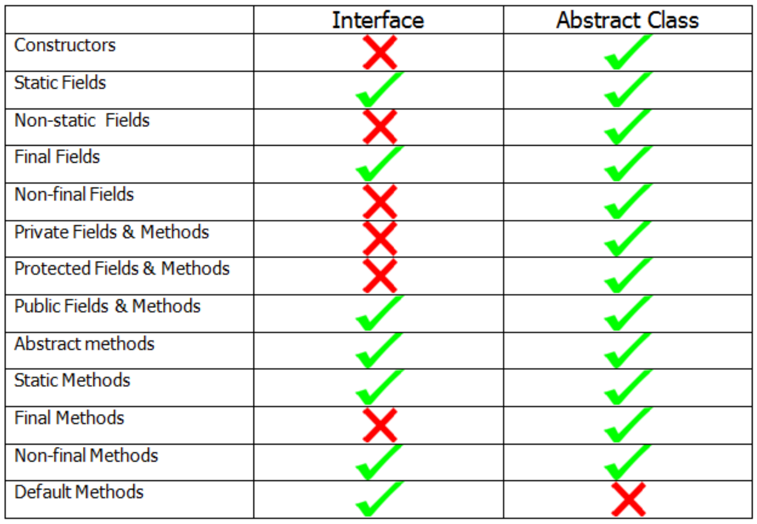
Methods in the interface should not have a body unless they are either static, a member of the interface, not the implementing class or its objects, or default. The default keyword before the return type of the method allows to provide an implementation in the interface which is common and shared for all implementing classes, however, they can override it if they want.

23. What is the Interface Segregation Principle (ISP)?

The Interface Segregation Principle (ISP) suggests that we should divide big, fat interfaces into smaller ones, each focusing on a single responsibility. Smaller interfaces are less likely to change. Changes to one capability will only affect a single interface and fewer classes that depend on that interface. An interface should only allow relevant behaviors to be seen so a class should not be forced to implement irrelevant behavior.

24. How do you compare interfaces and abstract classes?

Both are abstract concepts and we cannot instantiate them. Interfaces are contracts and should only have method declarations, however, in new versions of java in specific situations you can have static and default methods with implementation and final fields inside your interface. On the other hand, abstract classes are partially-implemented classes. We use them to share some common code across their subclasses. Non-abstract methods and non-final variables are allowed in abstract classes. Since we don't have multiple inheritances of classes in java, but a class can implement as many interfaces as it wants, so there some chances of simulating multiple inheritances situations, but we should be aware of the complexity we are adding by doing this.



	Interface	Abstract Class
Constructors	✗	✓
Static Fields	✓	✓
Non-static Fields	✗	✓
Final Fields	✓	✓
Non-final Fields	✗	✓
Private Fields & Methods	✗	✓
Protected Fields & Methods	✗	✓
Public Fields & Methods	✓	✓
Abstract methods	✓	✓
Static Methods	✓	✓
Final Methods	✗	✓
Non-final Methods	✓	✓
Default Methods	✓	✗

25. Do we have multiple inheritance? What is the Diamond problem?

Multiple inheritance as extending multiple classes is not allowed in Java, however, implementing multiple interfaces is possible. The diamond problem happens when we have multiple inheritances in a way that a class is extending 2 other classes and those parent classes are at least children of the Object class. In this case, if there is a common method in parent classes, like overriding toString() function from the Object class, it's not clear that which method should be inherited by the child class.

After Java 8, some new possibilities like default methods were introduced about interfaces. They are allowed to have the body to share the common code with classes implementing them. Knowing that each class can implement multiple interfaces, so a similar diamond problem might happen here and to prevent this, there is rule that in this case the class must have its own body for that specific method coming from 2 interfaces. The same approach could be used for classes, but so far, we have this only for interfaces.

26. What is UML and how it's being used?

The Unified Model Language can be categorized and described as:

- Behavioral – useful for web application design
    - Use Case Diagram: a high-level view from outside of the system which models the interaction between things outside (actors) and functionality within the system (use cases)
    - Activity Diagram: a low-level view of the system's logical flow with an initial and final node
    - Sequence Diagram: models a single scenario and shows the interaction between classes involved
- Structural – useful for Object-Oriented Programming overview which is modular
    - Class Diagram: models classes and interfaces within OOP program, and arrows representing relationships as:
        - Inheritance connectors
            - Generalization: IS-A relation, class from class
            - Realization: Does-A relation, class from interface
        - Dependency connectors
            - Dependency: Uses-A relation

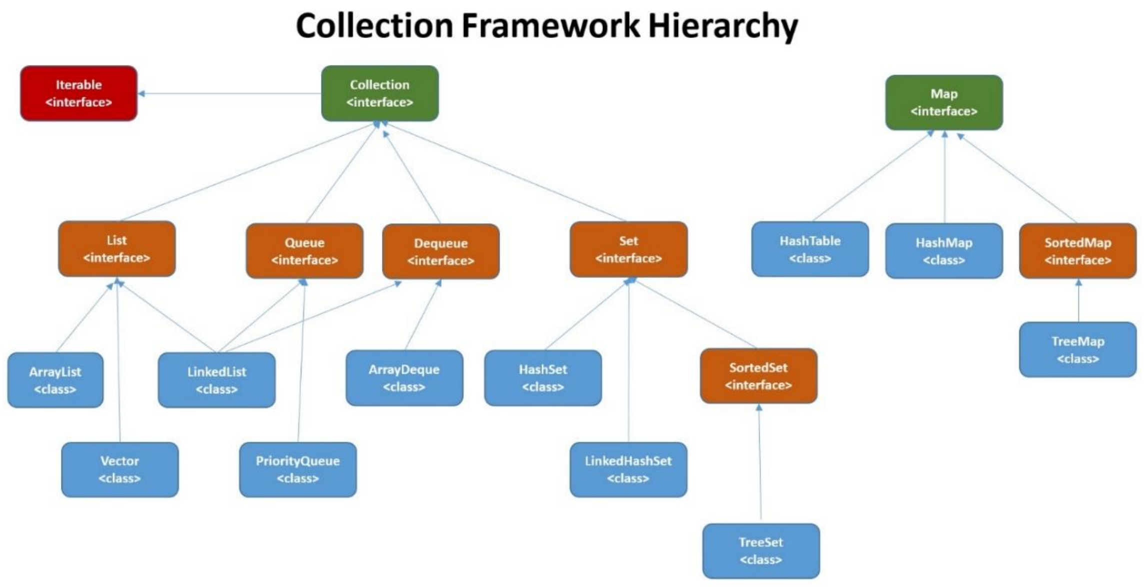
- Association: Has-A relation
- Aggregation: Has-Many relation but can be independent
- Composition: Has-Many relation but cannot be dependent

27. What is the difference between Collection and Collections?

The Collection is the root interface of grouping variable types hierarchy and defines only basic functionality like add() and remove() and itself extends the Iterable interface. Different sub-categories of this interface might be ordered, sorted, duplicate accepted, or not.

On the other hand, Collections is a class with many static methods which does different operations like sorting on the classes implementing collection interface. They both are provided in the Java Collections Framework as part of its API.

28. What do you know about different collections?



Unlike arrays, all sorts of collections are resizable. Any of the LinkedCollections are providing this feature using references to the next object of the collections, and other types start with an initial capacity, and when it gets full, allocates a new space in the memory with more capacity and copies all existing elements in the new collection. In different scenarios, either of them is faster and there is no general rule.

The ordered collections, like PriorityQueue, need the comparison to keep the order, so the object we are grouping by this type should implement the Comparable interface. Also, the no duplicate collections like sets, use the hashCode() and equals() methods as part of their implementations.

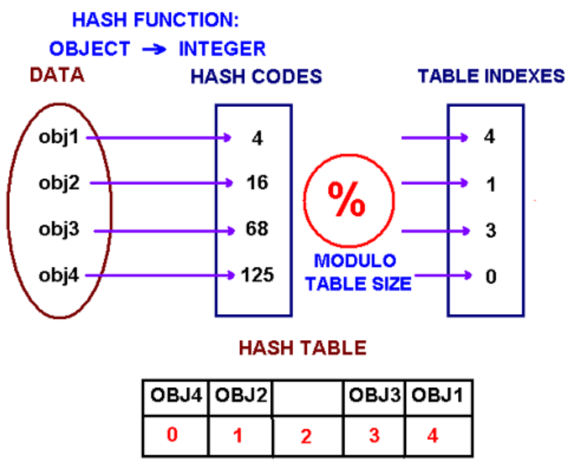
Among Lists, ArrayList and Vector are similar, but Vector is slower because it is thread-safe. In addition, when the initial capacity of ArrayList reaches, it increases the size by 50%, while for the Vector it's 100%. Among the set collections, HashSet stores data in a HashTable, but TreeSet stores them in a tree (ordered) so it is slower.

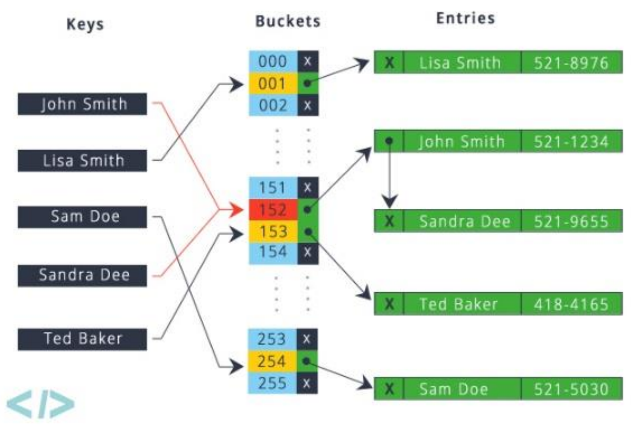
Maps are another type of grouping which maps a key to a value. No duplicates are allowed in keys, and there is one and only one value per key. The insertion order is not kept in them necessarily (Like HashSet). One null value is allowed for the keys of HashMap but not for HashTable. And HashTable is thread-safe.

29. How does hashing work? What is Collision? How does it get handled?

In the hashing process, first, it converts the object (based on attributes we specify in the hashCode() method) to a hash code. Then it reduces the size or value of the hash code using a modulo operation and determines an index for the object. There is one bucket for each index, and the objects with the same index get stored in the same bucket with a linked node algorithm. This will fasten the searching process considerably that we say it changes from O(n) to O(1).

If 2 objects are equal, the hash code and index will be equal, too. But the reverse conclusion is not true always. When 2 different objects resulted in the same hash code or same index, it is called a collision. It might increase the searching time complexity but no chance of error exists. When searching for an element in a hashed collection, it converts the element to the index, goes into the related bucket, and compares all the nodes there with the target item. That's why it never goes wrong.





30. What are Comparable and Comparator? What is the difference?

Any class that we want to use a sort function on it or be stored in a sorted type of collection, must implement the Comparable interface, then consequently inside the class override the compareTo() method with the related implementation returning a negative, zero, or positive value.

Another way of sorting an object is to pass a collection and a comparator object to one of the overloaded sort functions. This comparator should be an object of a class implementing the Comparator interface, then overriding the Compare() method which takes 2 objects of our class, based on any logic on its fields that we want, returns a negative, zero, or a positive number.

31. How do you compare Stack and Queue concepts?

Queue collections follow the FIFO pattern, which stands for First In First Out. An example is the files you send for a central printer in an office, they will get printed in the same order that the printer received them. In java, we don't have a queue class, but as an example, LinkedList is a class implementing queue interface and we can use them for the queue concepts, like dequeue and enqueue.

On the other hand, the Stack collection follows the LIFO pattern, Last In First Out. There is a stack class, and you can have an object of them and use push and pop methods. An example of stack usage is when you want to provide the Undo function for your website. Any time you need to undo, the last URL you entered in your stack should come out.

32. How do you compare ArrayList and LinkedList?

They are both implementing Iterable, Collection, and List interfaces. ArrayLists are considered as resizable arrays. Under the hood, at each time they are a fixed-size array, but when they get full, it finds another space in the memory to store a new fixed-size array with 50% more capacity. It copies all the current elements into the new array, then, you can continue to add new elements. Searching through the ArrayList elements is O(n), as well as Insertion and Deletion because it should move all other elements after that index. But searching by an index is fast because it just adds the index to the address of the first element and finds the address of the element at your desired index.

The LinkedList concept is different. Each node in a LinkedList stores a value and a pointer to the address of the next node. So it takes more memory for each element but there is no fixed size even in the background. It never needs to copy current nodes into a new address. Searching for a value again is O(n), but searching for an index is not fast because it has to start from the first node, and goes to the next node one by one. Instead, when it found your desired node, for Insertion and Deletion, it just needs the change the pointer in 1 or 2 nodes. There is no general rule to say one of them is always better.

33. In what classes we can use the for-each loop?

Any class, usually collections, that by default is implementing Iterable interface can be used to get an iterator from and work with for-each loops. If we have our own class, the class should implement the Iterable interface and then override the Iterator method to return an iterator object specifically for that class. This class should be a nested class implementing the Iterator interface which has 2 overridden methods: hasNext() and Next().

34. What are Generics? What is Type Erasure?

In cases that we want to create a class or a function to handle different types of input variables, instead of overloading our function with different signatures or repeat our code for different types of classes, we can use generics to code once but deal with all our desired objects dynamically. It also ensures compile-time type safety and helps detect bugs earlier. Casting and type checks will decrease but might be needed sometimes. All collections are defined using generics. If we want to use generics inside a function, either the function itself should be declared as generic or the class which includes this function should be defined as generic.

However, when running, java converts our generics into simple object type in its bytecode and this is called Type Erasure. The other type of this erasure is where we want to constraint our generic. We restrict out generic as accepting only objects with some specific features, but the java compiler when creating the bytecode only takes one of these classes. For example, substitutes the generic with only Object when we don’t have any constraints.

35. How can we add more specifications to our generic?

To restrict the generic, we should use the extends keyword when declaring the generic which means that it should only accept types that extending or implementing this specific class or interface like Number or Closable. We also can have multiple constraints like extending comparable and cloneable both.

The ? wildcard gives us more flexibility. For example, ClassName<? extends Vehicle> accepts all subclasses of the Vehicle and even itself. The super keyword, like ClassName<? super Car>, makes the generic accept all parents of that class up to the Object, and itself.

36. Does a generic accept a primitive type variable?

Indirectly yes but directly no. In fact, each of the primitive type variables like int and char has a wrapper class like Integer and Character. Whenever we pass a primitive type to a generic, it automatically converts it into its wrapper class object, which is called boxing, and passes it as a referenced type that is accepted by generics. Similarly, when we are putting a generic return of a function to a primitive variable, it automatically makes it a primitive type, which is also known as unboxing.

However, if the return type of a method is defined in an interface as a generic, when overriding the method in our class, in the method signature we must use the wrapper class name, even if inside the function we will be returning a primitive type. That will be handled by auto-boxing.

37. What is VarArg in java?

When we want to build a function that might take any number of parameters (from the same type) instead of overloading too many situations, we can use VarArgs (ObjectType… ObjectArrayName). These parameters will be saved like in an array inside the function. Only 1 VarArg per function is permitted and it must be the last parameter in the method signature.

38. What is Casting?

Casting is converting objects to their parent or child object. Upcasting is assigning a child object to a parent variable. It is happening always implicitly and is safe, however, the methods and field from the parent class will be visible only. Downcasting is the reverse operation and needs explicit casting using parenthesis before the parent object which is going to be assigned to a child variable. Since a problem here might not be detected at compile-time, using the instanceOf function for the parent object enables ensuring the correct downcasting.

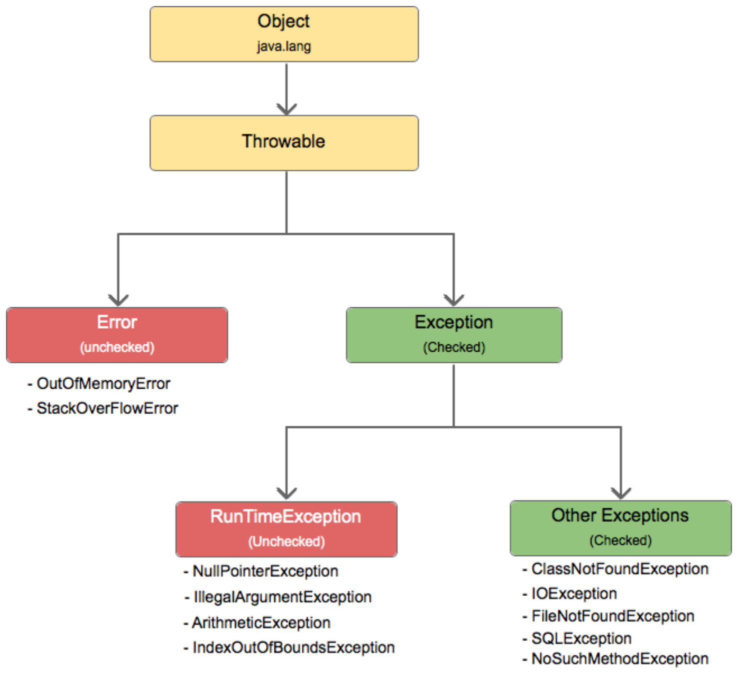
39. What is Exception and what are different types in Java?

An exception is a disruption in the normal flow of the instruction during the program's execution. Anytime an exception is thrown, the runtime system searches the call stack for a block of code that can handle it. Any method that doesn't handle it, is popped off the stack even if has not finished yet. Exception handling can help to prevent the program from crashing by specifying which method, to how to handle each situation outside of our code's control. Totally, we have 3 types of exceptions:

- Checked exceptions: compile-time exceptions we should anticipate and the java compiler forces us to handle them explicitly using a try-catch block or using throws keyword to declare that the method might throw that and caller must handle it.
- Unchecked exceptions: runtime exceptions are caused by programming errors or unvalidated wrong user inputs. The compiler doesn't notice them before running the application. We should prevent these exceptions from happening by thoughtful defensive coding and enough testing, especially for edge cases. Adequate if statements help.
- Errors: unrecoverable exceptions external to our application like when JVM is running out of memory. Programming mistakes like infinite loops or recursions, or outside problems like losing connection with the database could cause this type of exception. They should not be caught.

40. How does the order of catching exceptions matter?

When catching multiple exceptions we should use subclass exception at the top and the base class exception at the bottom. Otherwise, the parent exception catches all of them and non of your other catch blocks work.



41. What does the Finally keyword do?

That's a block we put it after our last catch block and it will get executed no matter whether if an exception was thrown and went through one of the catch blocks or nothing was wrong with the code. It's something to make sure that those lines will be executed and is a good place to close some objects and clean up resources. It might not get executed if JVM exits before or the thread is interrupted or killed.

42. What is the try-with-resources statement?

If we are going to initialize an object inside of our try block and we want to make sure that it will get closed after the block, instead of doing this manually in the finally block, we can put the deceleration of the variable inside the parentheses after try and before curly braces so it will get closed automatically. But that class must implement the AutoCloseable interface. As a result, objects remain scoped to the try block.

43. How can we throw a custom exception?

To create a custom Exception, we just need to create a class with that name and extend either the Exception or RuntimeException class. In both ways, it is extending the Throwable interface so then we can use the throw keyword for a new object of this class. We also can define constructors for our custom exception class and pass its string message to its superclass constructor.

44. What is the chaining exception?

There are times that we want to wrap one or some of our exceptions in a more general exception. So when the small exception should be thrown, we can throw the general exception while passing the small exception as it's constructor throwable cause argument or we can make the smaller one as the initCause of the general exception and then throw it. In this case, we just need to handle the general exception and show a general message to the user, however, if we print the stack trace of the exception get caught, we can see the cause exception.

45. What are SOLID principles?

They are a set of principles helping to prevent rigid, fragile, immobile, repetitive, and over-designed code and increasing reusability, scalability, extensibility, and maintainability.

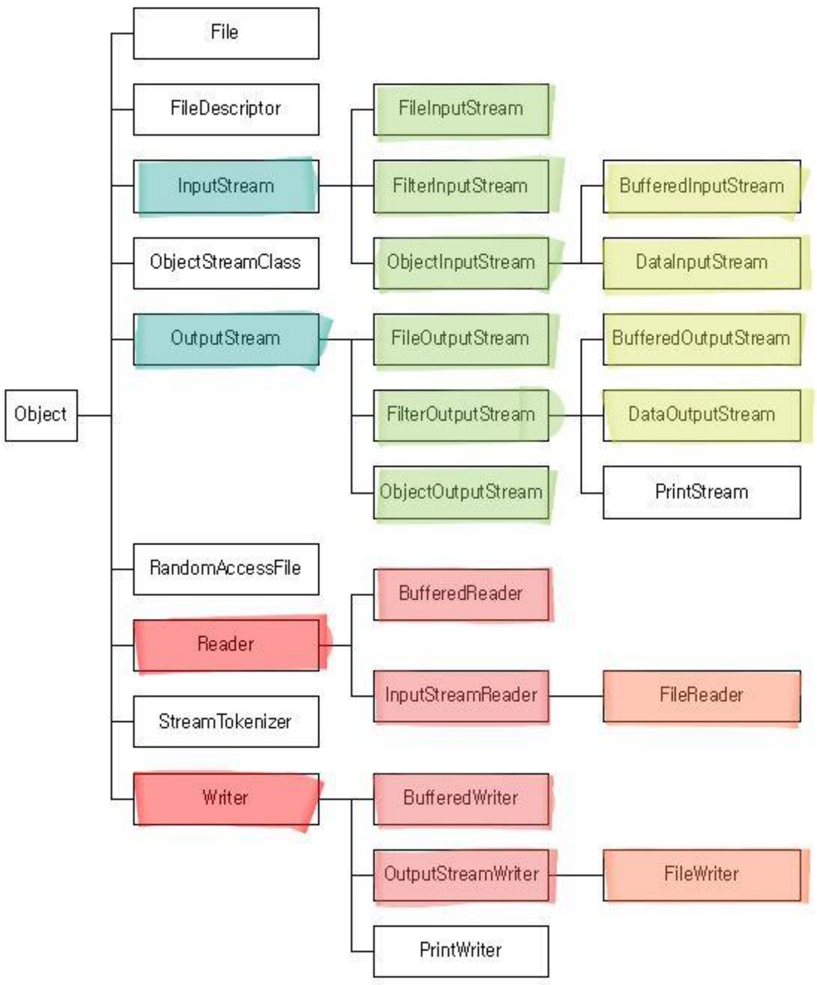
- Single Responsibility (SRP): A class should have only 1 reason to change (each should perform a set of closely related tasks)
- Open Closed (OCP): classes and functions should be open for extension (new behavior) but closed for modification (without changing source code)
- Liskov Substitution (LSP): subclasses should be substitutable for their base classes which means they should correctly fulfill expected behaviors inherited from super classes.
- Interface Segregation (ISP): interfaces should be small, each dealing with one aspect of a problem
- Dependency Inversion (DIP): classes (details) should depend upon abstract concepts, rather than concrete implementations.

46. What is a Stream and how they are represented in the java.io package?

A stream is a flow of data from a writer source at one end, input, and a reader destination at the other, output. In the java.io package, they are represented by 4 abstract classes where InputStream and OutputStream are based on bytes (low level) but Reader and Writer are based on characters.

InputStream and OutputStraem define basic functionalities for reading and writing bytes. Some of the common objects of these classes are System.in, System.out, and System.err created by JVM. Two of their children, FileInputStream and FileOutputStream read and write raw bytes from/to a file and DataInputStream and DataOutputStream read/write any primitive except char.

Reader and Writer are also providing basic functionality when working with characters and support Unicode. InputStreamReader and OutputStreamWriter are 2 subclasses that InputStream and OutputStream must be provided in their constructors and they can work with files. FileReader and FileWriter are subclasses of them which can accept a file directly. Buffered streams provide buffering, increasing efficiency by reducing the number of physical read/write operations. They can read/write larger blocks of characters at the same time.



47. What is Concurrency and how does it work?

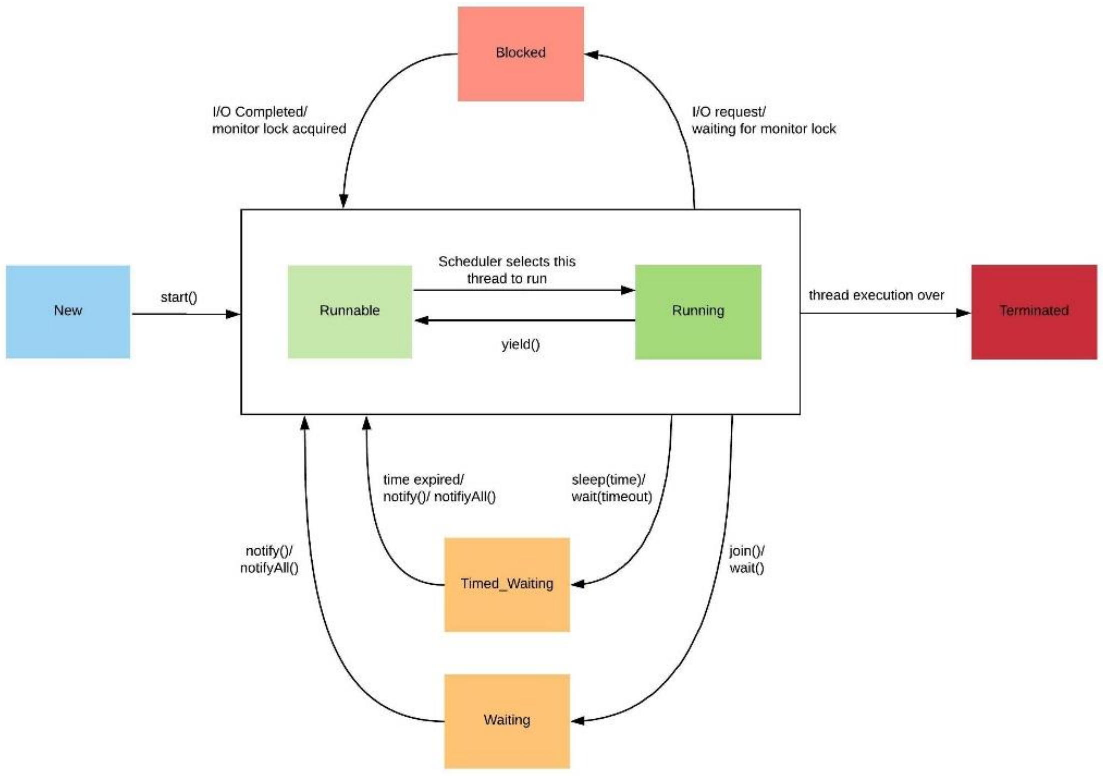
Concurrency is the ability of a program to perform more than one thing at once. However, running different things on a single core thread of CPU at the same time does not happen in reality. It generally works through Context Switching. CPU can perform a number of operations per second, and these are split up between multiple programs or tasks by the Scheduler. Each time switching the program that gets the chance on the CPU includes some savings and new loadings so is costly.

48. What are Process, Thread, and Multithreading?

A process usually refers to one program which gets its own allocated memory and resources. Each process has at least one thread or might have several, which are also called lightweight processes. They share the memory and resources with other threads in a way that there is only 1 common heap for all threads but they, each, have one stack. A Thread refers to a sequence of instructions executed line by line and Multithreading is when a single program has more than one thread. Multithreading can 1) Utilise multiple cores, 2) Improve responsiveness, 3) Deal with several tasks at once, and 4) Perform background tasks.

49. How does threading work in Java?

In Java, every application has at least 1 thread which is the main thread, and starts running when the main() is invoked. We can also create other threads to run concurrently. The thread scheduler decides which one runs when. Each thread has different states shown in the picture.


Blocked
I/O Completed/ monitor lock acquired
I/O request/ waiting for monitor lock
New
start()
Runnable
Scheduler selects this thread to run
Running
yield()
thread execution over
Terminated
time expired/ notify()/ notifiyAll()
sleep(time)/ wait(timeout)
notify()/ notifyAll()
join()/ wait()
Timed_Waiting
Waiting

50. How can we create new threads in Java?

- Create a class, extend the Thread class, and override the run() method, and call the start() method on an object of this class. Since there is a limitation on extending other classes, this might make system design issues.
- Create a class, implement the Runnable interface, override the run() method, pass an object of this class to an object of the Thread class as its Runnable and call the start on that object. A call to the start() function of any thread, calls the run() method itself.

51. What is JavaDoc and what are important annotations?

JavaDoc is a form of commenting that can be used to generate API-like HTML documentation. It will result in no need of looking at code when using a library.

- /** JavaDoc comment */
- @author, @version → start of the class
- @param, @return, @throws → before a method
- @see, @since, @deprecated

52. What are JavaBean conventions?

- The public, no-argument constructor

- Properties with getter and setter

  Implementing Serializable

53. What is Serialization?

It’s the process of converting java object state to byte arrays resulting in data that can be stored or transmitted. The problem with them is that they can only be parsed (deserialized) by other Java applications, not other languages and platforms.

54. What is JSON?

JSON, standing for JavaScript Object Notation, is a lightweight human and computer-parsable file format which is based on JavaScript syntax but is language-independent and can be used to communicate between applications. It is constructed based on Objects in {} and Arrays in [] including values of String, Number, Booleans, and null values all in “ ”, both in a comma-separated list. In an Object name and value pairs are connected with a : in between.

55. What is Jackson?

It’s a project that includes libraries to read and write JSON and other data formats providing 2 functionality: 1) Data-binding: defines how Java objects will be represented in JSON, and 2) Streaming: the reading/writing JSON from/to various applications. ObjectMapper is the class providing these via 2 overloaded methods of writeValue() to serialize Java objects and readValue() to deserialize JSON. By default, Jackson assumes that our Java code follows the JavaBean convention (with the exception that doesn’t need to implement Serializable) so it deserializes objects via public no-arg constructors and defines the attribute names based on only getters and setters. In addition, there are annotations help to customize these processes by Jackson:

- @JsonProperty(“property”): Associates the field/getter/setter with a specific JSON property name
- @JsonCreator: Defines a specific constructor to deserialize with
- @JasonInclude: Customizing inclusion/exclusion of properties when serializing
- @JsonIgnoreProperties: providing a list of properties to ignore before the class
- @JsonIgnore: To ignore just the following field/getter/setter