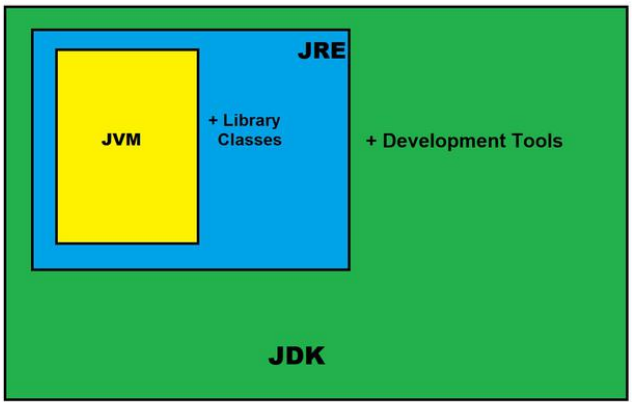
Technical Interview Questions

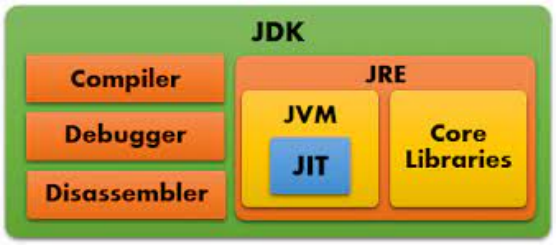
Core Java

1. What is JDK? And name some components of JDK.
   JDK (Java Developer Kit) contains tools which a Java developer needed to develop the Java programs, and **JRE** to run the program. In simple terms we can say that ***JDK = JRE + Development Tools***.

   The tools include *Java Archive* ***(jar)***, *Java Compiler* ***(javac)***, *Java Disassembler* ***(Javap)***, *Java Debugger* ***(jdb)***, *Java HeaderFile Generator* ***(javah)***, *Documentation* ***(javadoc)*** and many others.

2. Difference between JVM, JRE, and JDK





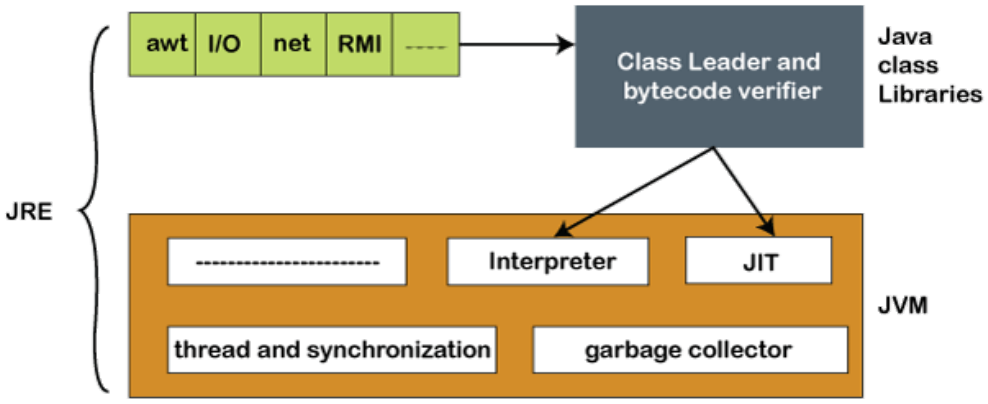
JVM = Class Loader Subsystem + Runtime Data Area + Execution Engine

JRE = JVM + Libraries

JDK = JRE + Development Tools

3. Name some popular JDKs
   - Oracle JDK
   - OpenJDK
   - IBM J9 JDK

4. What are some of the components of JRE?

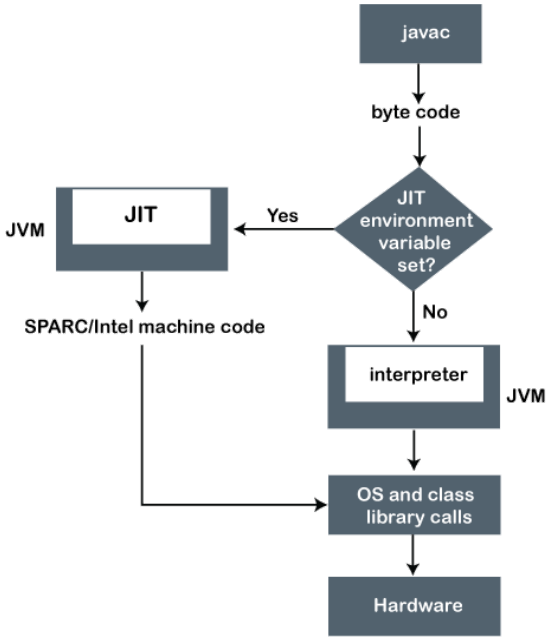


Functional Relationship of the JIT to the JRE and JVM

5. What is JIT compiler in Java?
   The Just-In-Time (JIT) compiler is a component of the Java Runtime Environment that improves the performance of Java programs by compiling bytecodes into native machine code at run time.

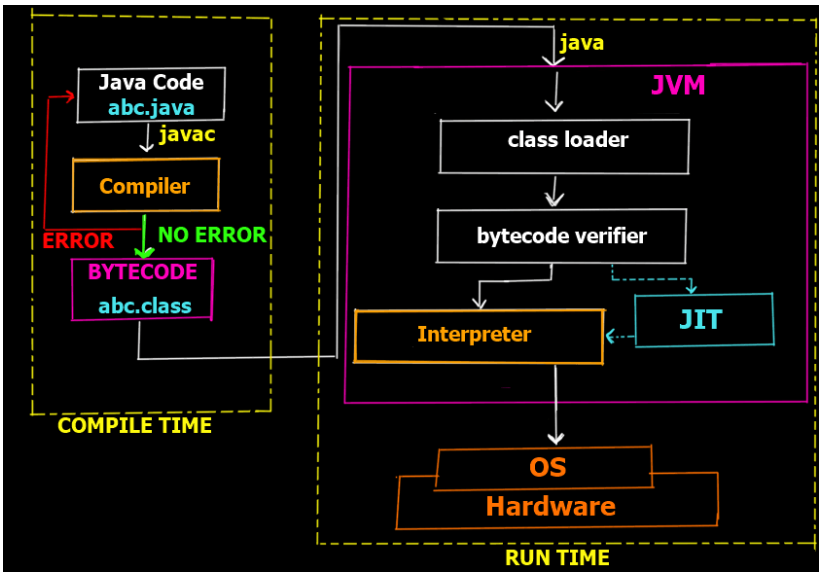
6. What is the difference between JIT compiler and a traditional compiler?
   A traditional compiler will compile all code to a machine language before a program starts to run. JIT compilers generate code while the program is running.

7. What is the [compilation process] in Java?

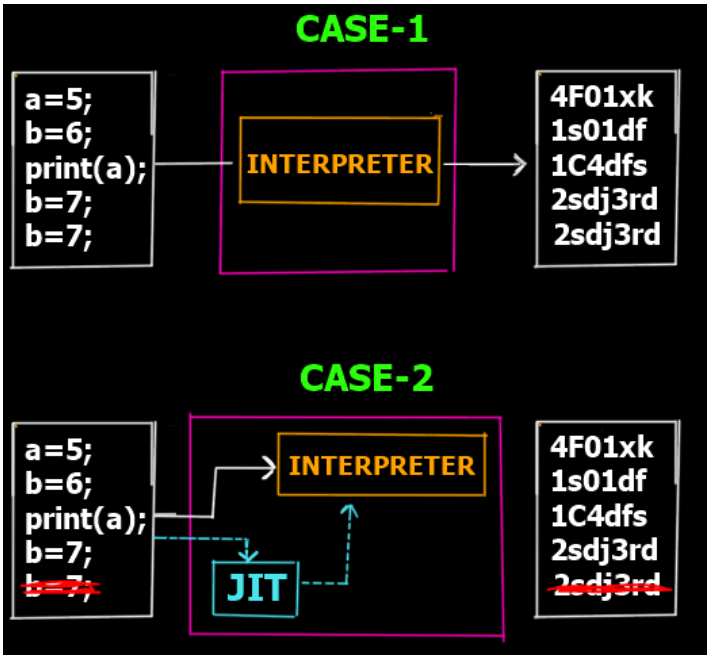


JIT Compilation Process

8. Execution process of Java programs



9. Working of JIT compiler



- **Case-1:** Lets assume we have 5 lines which are supposed to be interpretted to their corresponding machine code lines. So as you can see in the Case-1 there is no JIT involved. thus the interpreter converts each line into its corresponding machine code line. However if you notice the last 2 lines are the same (consider it a redundant line inserted by mistake). Clearly that line is redundant and does not have any effect on the actual output but yet since the interpreter works line by line it still creates 5 lines of machine code for 5 lines of the bytecode.

- **Case-2:** In case 2 we have the JIT compiler. Now before the bytecode is passed onto the interpreter for conversion to machine code, the JIT compiler scans the full code to see if it can be optimized. As it finds the last line is redundant it removes it from the bytecode and passes only 4 lines to the interpreter thus making it more efficient and faster as the interpreter now has 1 line less to interpret.

10. What are some optimizations done by JIT compiler?
    - Method In-lining
    - Local Optimizations
    - Control Flow Optimizations
    - Constant Folding
    - Dead Code Elimination
    - Global Optimizations

11. What is coupling in Java?
    Coupling is nothing but the dependency of one class on the other. (In coupling, two classes or objects collaborate and work with each other to complete a pre-defined task.)

12. What are the types of coupling?
    a. Loose coupling (Example: creating an object of a class **BankAccount** in the `main()` method of the driver class **BankOperations**)
    b. Tight coupling (Example: having the object reference of a class **Address** as a data member of another class **Order**)

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

13. Why does coupling matter?
    If a class changes its behavior in the next release of the application program, all the classes that depend on it may also be affected. In this situation, we will need to update all the coupled classes.
    Therefore, if the coupling between classes will be low, we can easily manage them. Thus, we must remove unnecessary coupling between classes.

14. What is cohesion in Java?
    Cohesion is making sure that a class is designed with a single, well-focused purpose. In object-oriented design, cohesion refers all to how a single class is designed. The more focused a class is, the more cohesiveness of that class is more.

15. Give examples of the two types of cohesion
    Suppose we have a class that multiplies two numbers, but the same class creates a pop-up window displaying the result. This is an example of a low cohesive class because the window and the multiplication operation don't have much in common.

    To make it high cohesive, we would have to create a class Display and a class Multiply. The Display will call Multiply's method to get the result and display it. This way to develop a high cohesive solution.

16. What is the difference between cohesion and coupling?
    - Cohesion of a single module or component is the degree to which its responsibilities form a meaningful unit; higher cohesion is better.
    - Coupling between modules or components is their degree of mutual interdependence; lower coupling is better.

17. What are the different ways to create a new object in Java:
    a. Using `new` keyword
    b. Using `newInstance()` of `java.lang.Class`
    c. Using `newInstance()` of `java.lang.reflect.Constructor<T>`
    d. Invoking `clone()`
    e. By deserialization
    f. By factory method

18. What is Data hiding?
    Data hiding is the process in which the field is declared private within the class to hide so that no one can access it from outside the class.

(Concept involved in Data hiding: *getters* and *setters*, i.e. accessors and mutators)

19. How to implement Encapsulation in Java?
    There are two key points thereby we can achieve or implement encapsulation in Java program:
    - Declaring the instance variable of the class as private so that it cannot be accessed directly by anyone from outside the class.
    - Provide the public setter and getter methods in the class to set/modify the value of the variable.

20. What is meant by tightly encapsulated class in Java?
    If each variable is declared as private within the class, it is called tightly encapsulated class in Java.

21. Modifiers in Java for variables, functions, and classes:
    a. Accesss Modifiers
        i. `private`
        ii. `default`
        iii. `protected`
        iv. `public`
    b. Non-access Modifiers
        i. `static`
        ii. `final`
        iii. `abstract`
        iv. `synchronized`
        v. `transient`
        vi. `volatile`
        vii. `native`
        viii. `strictfp`

22. Why can constructors not be declared `static` in Java?
    In general, a `static` method means that "the method belongs to the class and not to any particular object". But a constructor is always invoked with respect to an object, so it makes no sense for a constructor to be `static`.

23. What is the use of `final` keyword in Java?
    - To create constant variables and parameters
    - To prevent method overriding
    - To prevent inheritance

24. What is blank or uninitialized `final` variable?
    A final variable that is not initialized at the time of declaration is known as blank final variable.

25. Can we initialize blank `final` variable?
    Yes, but only in constructor.

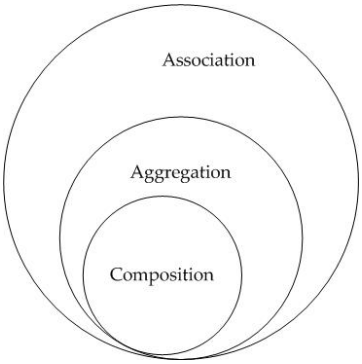
26. Can we initialize a `static` blank `final` variable?
    It can be initialized only in static block.

27. Can we declare a constructor `final`?
    No, because a constructor is never inherited.

28. What is the use of `volatile` keyword?
    The `volatile` keyword does not cache the value of a variable thread-locally and always read the variable from the "main memory". The `volatile` keyword cannot be used with classes or methods. However, it is used with **variables**. It also guarantees *visibility* and *ordering*. It prevents the compiler from the reordering of code.

29. What is the use of `strictfp` keyword?
    Java `strictfp` keyword ensures platform-independent floating-point computations. It ensures that you will get the same result on every platform if you perform operations in the floating-point variable.

30. Association, Aggregation, Composition (`HAS-A`) relationships
    - Association is a generalized concept of relations. It includes both Composition and Aggregation. (Example: An **User**'s Amazon account having a *List* of **Address**es)
    - Aggregation (The formation of a number of things into a cluster) differs from ordinary composition in that it does not imply ownership. In composition, when the owning object is destroyed, so are the contained objects. In aggregation, this is not necessarily true. (Example: An Amazon **Order** having a Delivery **Address**)
    - Composition (mixture) is a way to wrap simple objects or data types into a single unit. Compositions are a critical building block of many basic data structures. (Example: A **University** having a collection of **College**s)

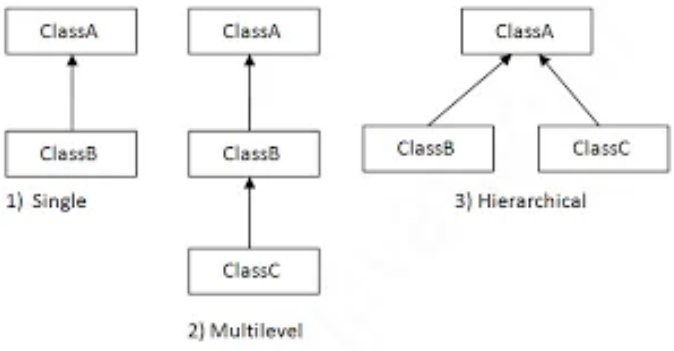


In both aggregation and composition, the object of one class "owns" the object of another class. But there is a subtle difference. In Composition the object of class that is owned by the object of its owning class cannot live on it's own(Also called "death relationship"). It will always live as a part of it's owning object whereas in Aggregation the dependent object is standalone and can exist even if the object of owning class is dead. So in composition if the owning object is garbage collected the owned object will also be which is not the case in aggregation.

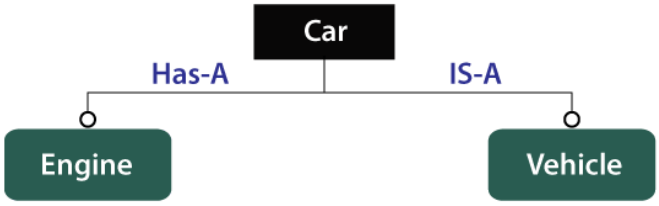
Trick to remember the difference :

- "Has-**A**": **A**ggregation
- "**P**art-**O**f": c**O**m**P**ositoin
- "**I**s-a": **I**nheritance

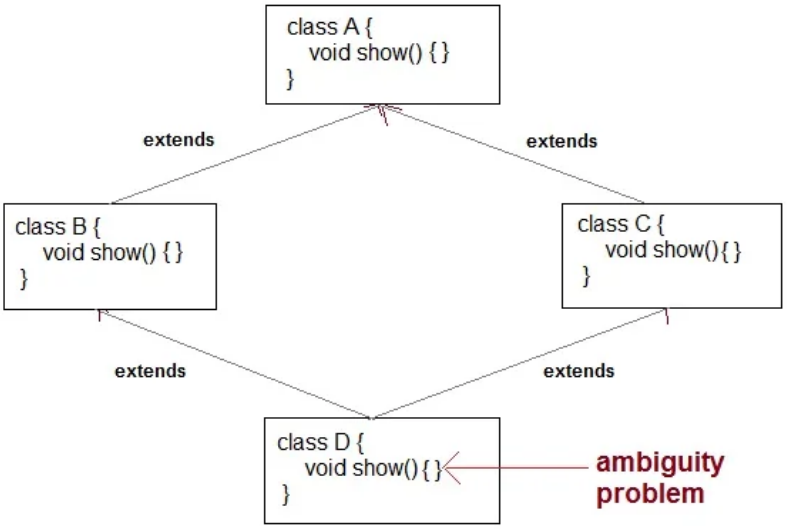
31. Inheritance (`IS-A`) relationship



32. Give an example of a class using both `HAS-A` and `IS-A` relationship.



33. Why multiple inheritance is not supported in Java?
    - To remove ambiguity.
    - To provide more maintainable and clear design.



34. What is the use of `super` keyword in Java?
    `super` keyword is a reference variable that refers to an immediate superclass object. It is used for the following purposes:
    A. To refer immediate parent class instance variable.
    B. To call immediate parent class constructor.
    C. To invoke immediate superclass method.

35. Why do we use this keyword in Java?
    `this` keyword is a reference variable that refers to the current class object. It holds the reference to current class object or same class object.

    There are six usages of Java this keyword:
    A. `this` reference can be used to refer to the current class instance variable.
    B. `this` keyword is used to call the non-static method of the current class.
    C. `this()` can be used to invoke the current class constructor.
    D. `this` keyword can be used as a parameter in the method call.
    E. The keyword "`this`" can be used as a parameter in the constructor call.
    F. It can also be used to return the object of the current class from the method.

36. Can you use both `this()` and `super()` in a Constructor?
    NO, because both `super()` and `this()` must be first statement inside a constructor. Hence we cannot use them together.

37. What is Binding in Java?
    The connecting (linking) between a method call and method definition is called binding in Java.

38. What is the difference between Static binding and Dynamic binding in Java?
    Static binding in Java occurs during compile time while dynamic binding occurs during runtime. In simpler terms, Static binding means when the type of object which is invoking the method is determined at compile time by the compiler. While Dynamic binding means when the type of object which is invoking the method is determined at run time by the compiler.

Static Binding	Dynamic Binding
It is resolved at compile time	It is resolved at run time
static binding use type of the class and fields	Dynamic binding uses object to resolve binding
Overloading is an example of static binding	Method overriding is the example of Dynamic binding
private, final and static methods and variables uses static binding	Virtual methods use dynamic binding

39. What are the different ways of method overloading in Java?
    Method overloading can be done by changing:
    a. The number of parameters in two methods.
    b. The data types of the parameters of methods.
    c. The Order of the parameters of methods.

40. Rules of method overloading:
    Two methods will be treated as overloaded if both follow the mandatory rules below:
    a. Both must have the same method name.
    b. Both must have different argument lists.

    And if both methods follow the above mandatory rules, then they may or may not:
    - Have different return types.
    - Have different access modifiers.
    - Throw different checked or unchecked exceptions.

41. How does the Java compiler differentiate between methods in Compile time Polymorphism?
    During compilation, Java compiler differentiates multiple methods having the same name by their signatures.

42. Is it possible to implement runtime polymorphism by data members in Java?
    No, we cannot implement runtime polymorphism by data members in Java.

43. What are the rules of covariant return types in Java?
    There are mainly three rules for covariant return types:
    - The return type of overriding method in the subclass should be either the **same** as the return type of the overridden method or a **subclass** of its return type.
    - The return type of overriding method in the subclass should not be a parent of the overridden method return type.
    - The covariant return type is applicable only for object types not for primitive types.

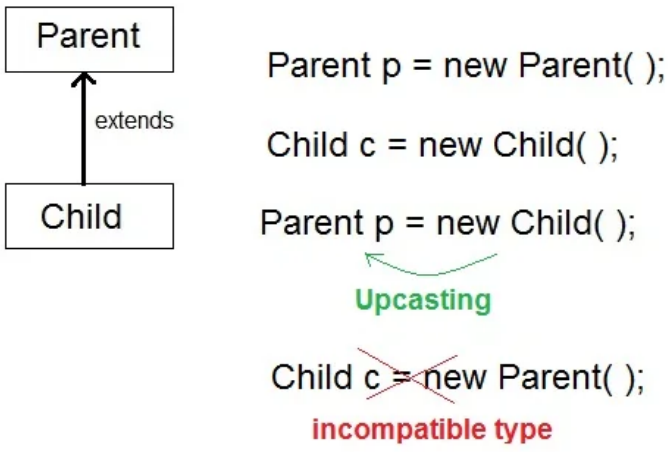
44. Rules of method overriding:
    a. There must be an IS-A relationship between classes (inheritance).
    b. Method signature (name and parameter list) must be the same for both parent and child classes.
    c. Access modifier of child method must not restrictive than parent class method.
    d. Private, final and static methods cannot be overridden.

45. Difference between method overloading and method overriding

Method Overloading	Method Overriding
Parameter must be different, and name must be same.	Both name and parameter must be same.
Compile time polymorphism.	Runtime polymorphism.
Increases readability of code.	Increases reusability of code.
Access specifier can be changed.	Access specifier cannot be more restrictive than original method (can be less restrictive).
It is also known as compile-time polymorphism or static polymorphism or early binding.	It is runtime polymorphism or dynamic polymorphism or late binding.
It is performed within a class	It is performed between two classes using inheritance relation.

It should have methods with the same name but a different signature.	It should have methods with same name and signature.
It cannot have the same return type.	It should always have the same return type.
It can be performed using static methods	It cannot be performed using static methods
It uses static binding	It uses the dynamic binding.
Access modifiers and non-access modifiers can be changed.	Access modifiers and non-access modifiers can not be changed.
It is code refinement technique.	It is a code replacement technique.
private, static, final methods can be overloaded	private, static, final methods cannot be overloaded
No restriction in throws clause.	Restriction in throws clause for checked exceptions only.
Example: *class OverloadingDemo{* *static int add1(int x, int y){return x + y;}* *static int add1(int x, int y, int z){return x + y + z;}* *}*	**Example:** *class Base {* *void a() {System.out.println("A");}* *}* *class Derived extends Base {* *void a() {System.out.println("B");}* *}*

46. Explain Runtime Polymorphism or Dynamic method dispatch
Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime. This is how java implements runtime polymorphism. When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to. In simple words the type of object which it referred determines which version of overridden method will be called.



47. What is upcasting in Java?
    When Parent class reference variable refers to Child class object, it is known as Upcasting. In Java this can be done and is helpful in scenarios where multiple child classes extends one parent class. In those cases we can create a parent class reference and assign child class objects to it. (refer to the previous image)

48. What is downcasting in Java?
    This occurs when you cast the reference down the inheritance tree to a more specific class. (refer to the previous image)

49. What is a polymorphic reference?
    A polymorphic reference is a variable that can refer to different types of objects at different points in time. Any Java object that can pass the `IS-A` test can be considered polymorphic.
    Example:
    - `Parent obj = new Parent();`
    - `obj = new Child();`

50. Can we override `static` methods? Explain with reasons.
    No, we cannot override static methodd. Because s static method is bound to class whereas method overriding is associated with object i.e at runtime.

51. What is method hiding in Java?
    If a static method defined in the parent class is redefined in a child class, the child class's method hides the method defined in the parent class. This mechanism is called method hiding in Java or function hiding.

    It is an example of compile-time polymorphism. Compiler is responsible for method resolution based on reference type whereas, in method overriding, JVM is always responsible for method resolution based on runtime object.

52. Can we override the `start()` method in thread class?
    We can override `start()` method of `Thread` class because it is not final. But it is not recommended to override the `start()` method, otherwise it ruins multi-threading concept. Whenever we override `start()` method then our `start()` method will be executed just like a normal method call and new thread wont be created.

    (Extra info - We must call the `super.start()` method inside our overridden `start()` method to create a new thread)

53. What is a thread?
    In Java, the word "thread" means 2 different things:
    - An instance of class `java.lang.Thread`
    - A thread of execution

54. What is the difference between an a) instance of a Thread and a b) "Thread of execution"?
    A "Thread of execution" represents an individual process. An instance of a Thread is an object.

    A thread of execution is an individual process (a"lightweight") process that has its own call stack.

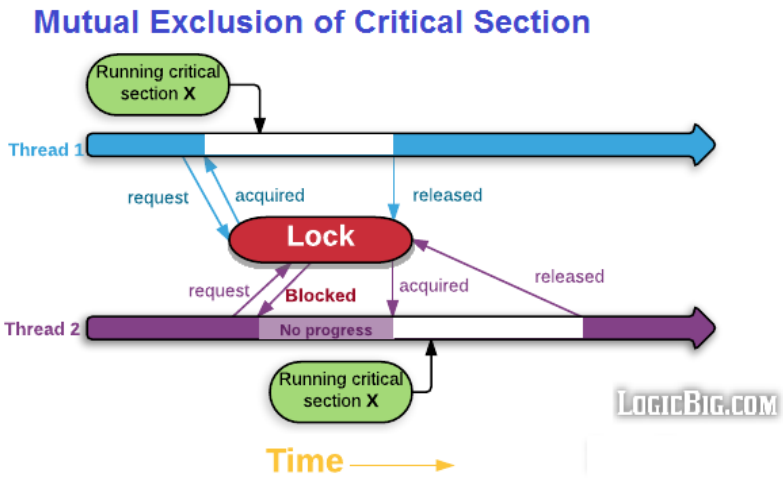
55. What are 2 ways you can instantiate a thread?
    - Extend the `java.lang.Thread` class.
    - Implement the `Runnable` interface

56. How are **intrinsic locks** used in thread synchronization in Java?
    An *intrinsic lock* (aka *monitor lock*) is an implicit internal entity associated with each instance of objects.

    The *intrinsic* lock enforces exclusive access to an object's state. Here 'access to an object' refers to an instance method call. Synchronization in Java is built around an internal entity known as the *intrinsic lock* or *monitor lock*.

    When a `synchronized method` is called from a thread or a `synchronized block` is being executed, it needs to acquire the *intrinsic* lock. The lock is released when the thread is done executing the method or an exception is thrown in the method which is not handled/caught.

    As long as a thread owns an *intrinsic* lock, no other thread can acquire the same lock. The other threads will block when they attempt to acquire the lock. The blocked threads will wait till the lock is released by the currently executing thread.



`Thread.sleep()` inside the synchronized method doesn't release the lock.

Intrinsic locks are reentrant. That means once a thread has acquired the lock on a method it doesn't need to acquired the lock on calling other method of the same object. The other method doesn't need to necessarily have `'synchronized'` keyword.

Constructors cannot be synchronized. Using the `synchronized` keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

A `static` method can have synchronized keyword. In this case, the thread acquires the intrinsic lock for the Class object associated with the class rather than an instance of the class.

57. Is intrinsic lock acquired on an object or a method?
    Intrinsic lock is on object, not on a method. As mentioned before, if a thread has acquired the lock, other threads will be blocked even if they are calling other `'synchronized'` methods of the same object. Non-synchronized methods won't be blocked.

58. What is reentrancy?
    Reentrancy allows the same thread to acquire the same lock again. Intrinsic locks **are** reentrant.

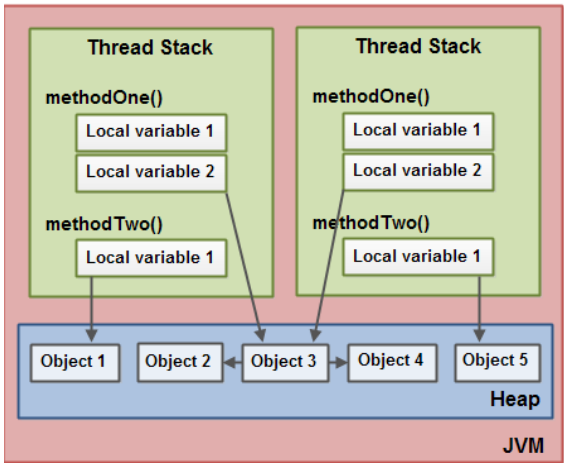
59. Difference between intrinsic locking & extrinsic locking?

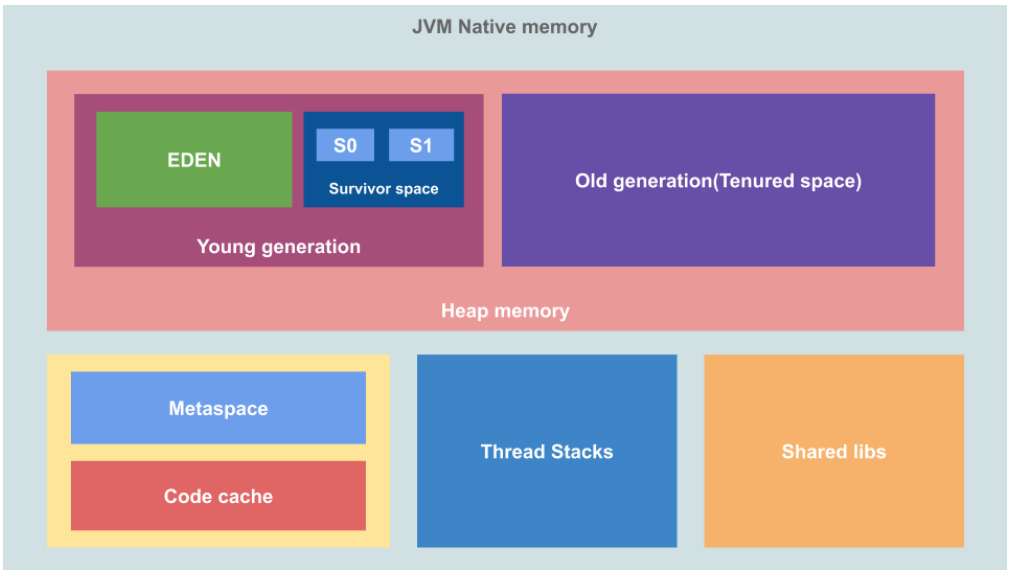
- Intrinsic locking: In Java, every object can **implicitly** act as a lock for purposes of synchronization; these built-in locks are called *intrinsic* locks. What is interesting with the term intrinsic is that the ownership of a lock is per thread and not per method invocation. That means that only one thread can hold the lock at a given time.
- Extrinsic locking: Extrinsic locking is, instead of using the built in mechanisms in `java.util.concurrent.locks` which gives you the utilities to specifically use explicit locks, e.g. `ConcurrentHashMap` . It is kind of a more sophisticate way of locking. There are many advantages (for example you can set priorities).

60. What is thread starvation?

    Thread starvation occurs when one or more threads are constantly blocked to access a shared resource in favor of other threads.

61. Java Memory Model





62. Singleton pattern
63. Comparators
64. Factory method
65. Calendar class
66. Instance of operator
67. In built final classes
68. Predefined classes in Java
69. Meaning of System.out.println();
70. Can we have constructor in abstract class and interface
71. Why is this used in synchronized block
72. Static keyword and can a class be defined as static
73. constructor chaining
74. Need of Collections framework
75. Locks in java
76. Static block usage
77. Why don't we use exception handling in `Scanner` class?
78. Foreach vs enhanced for loop
79. Assertion (testing of code using assert keyword)

80. Exception handling in Java
    Refer here for a quick overview of exception handling essentials.

    Refer to this cheat sheet for a thorough synopsis.

81. Purpose of `throws` keyword
    A function which declares `checked` exceptions with a `throws` clause is basically saying to the callee:
    "I'm throwing these exceptions, handle it yourself."

82. `throw` vs `throws`

throw	throws
Java **throw** keyword is used to explicitly throw an exception.	Java **throws** keyword is used to declare an exception.
Checked exception cannot be propagated using **throw**.	Checked exception can be propagated with **throws**.
If we see syntax wise, **throw** is followed by an instanceof Exception class Example : throw new NumberFormatException("The month entered, is invalid.");	If we see syntax wise, **throws** is followed by exception class names. Example : throws IOException,SQLException
The keyword **throw** is used inside method body.	**throws clause** is used in method declaration (signature).
By using **throw keyword** in java you cannot throw more than one exception. Example: throw new IOException("Connection failed!!")	By using **throws** you can declare multiple exceptions. Example: public void method() throws IOException,SQLException.

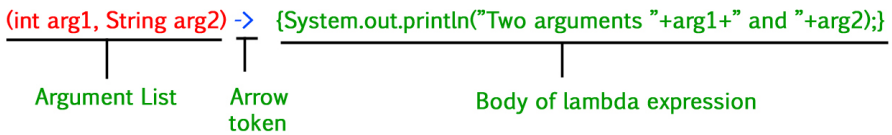
83. Generics in Java
    Refer to this cheat sheet for an overview.

84. Anonymous object - object which is not stored in a reference

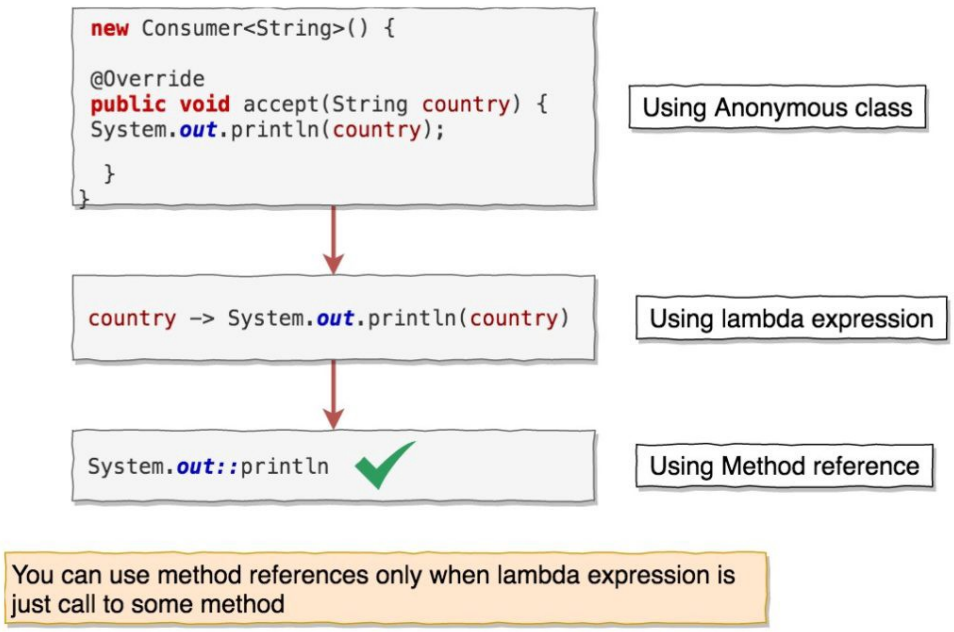
85. Types of Interfaces

86. Lambda Expressions
    Lambda expression facilitates functional programming, and simplifies the development a lot. A lambda expression is used to implement the SAM of a functional interface.

    Lambda expressions are added in Java 8 and provide below functionalities:
    - Enable to treat functionality as a method argument, or code as data.
    - A function that can be created without belonging to any class.
    - A lambda expression can be passed around as if it was an object and executed on demand.



87. What are method references?
    Method references are a special type of lambda expressions. They're often used to create simple lambda expressions by referencing existing methods.



88. What are the advantages of using method references?

```
Comparator<Person> byAge = Comparators.comparing(p -> p.getAge());

Comparator<Person> byAge = Comparators.comparing(Person::getAge);
```

Method reference

Readability

```
Collections.sort(people, comparing(Person::getAge));
```

Reusability

```
Collections.sort(people, comparing(Person::getAge).reverse());
```

Composability

```
Collections.sort(people, comparing(Person::getAge)
                         .compose(comparing(Person::getName)));
```

89. What are the different syntax/kinds of Java 8 method references?

Know all five kinds of method references

All are at times preferable to lambdas

Type	Example	Lambda Equivalent*
Static	`Integer::parseInt`	`str -> Integer.parseInt(str)`
Bound	`Instant.now()::isAfter`	`Instant then = Instant.now();` `t -> then.isAfter(t)`
Unbound	`String::toLowerCase`	`str -> str.toLowerCase()`
Class Constructor	`TreeMap<K,V>::new`	`() -> new TreeMap<K,V>()`
Array Constructor	`int[]::new`	`len -> new int[len]`

90. Explain the difference between the following statements:

```
1. Stream.of("x","y").forEach(System.out::println);
2. Stream.of("x","y","").filter(String::isEmpty);
```

Statement **(1)** is using a *bound receiver*. It will execute `println` on a specified instance of `PrintStream` - the `System.out` instance. Therefore, `System.out.println("x")` and `System.out.println("y")` will be executed as a result of passing that method reference to `forEach()`.

Statement **(2)** is using an *unbound receiver*. It will execute `isEmpty()` on each of the `String` instances of the Stream - i.e. `"x".isEmpty()`, `"y".isEmpty()` and `"".isEmpty()`.

91.

92. Collections in Java

93. Difference between a fail-fast and fail-safe iterator.
    - **Fail Fast Iterator**: Fail-Fast Iterators will throw an `ConcurrentModificationException` if you try to modify while iterating over it. Ex: Iterators returned by `ArrayList`, `HashMap`.
    - **Fail Safe Iterator**: Non-Fail Fast Iterators make a copy of the internal collection (object array) and iterates over the copied collection. Collections which come under "`java.util.concurrent`" package have fail-safe iterator. You can modify the collection while iterating over it. Ex: Iterators returned by `ConcurrentHashMap`, `CopyOnWriteArrayList`.

Parameters	Fail Fast Iterator	Fail Safe Iterator
Throw ConcurrentModification Exception	Yes, they throw CocurrentModificationExcepti-on if a collection is modified while iterating over it.	No, they do not throw any exception if a collection is modified while iterating over it.
Clone the Collection	No, they use original collection to traverse over the elements.	Yes, they use the copy of the original collection to traverse.
Memory Overhead	No, they don't require extra memory.	Yes, they require extra memory to clone the collection.
Examples	HashMap, Vector,ArrayList,HashSet	CopyOnWriteArrayList

94. Difference between HashMap, LinkedHashMap, and TreeMap.

Property	HashMap	TreeMap	LinkedHashMap
Iteration Order	no guaranteed order, will remain constant over time	sorted according to the natural ordering	insertion-order
Get / put / remove / containsKey	`O(1)`	`O(log(n))`	`O(1)`
Interfaces	Map	NavigableMap, Map, SortedMap	Map
Null values/keys	allowed	only values	allowed
Fail-fast behavior	Fail-fast behavior of an iterator cannot be guaranteed, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification	Fail-fast behavior of an iterator cannot be guaranteed, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification	Fail-fast behavior of an iterator cannot be guaranteed, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification

Implementation	buckets	Red-Black Tree	double-linked buckets
Is synchronized	implementation is not synchronized	implementation is not synchronized	implementation is not synchronized

95. Built-in Functional Interfaces in Java

Function Type	Method Signature	Input parameters	Returns	When to use?
Predicate<T>	boolean test(T t)	one	boolean	Use in conditional statements
Function<T, R>	R apply(T t)	one	Any type	Use when to perform some operation & get some result
Consumer<T>	void accept(T t)	one	Nothing	Use when nothing is to be returned
Supplier<R>	R get()	None	Any type	Use when something is to be returned without passing any input
BiPredicate<T, U>	boolean test(T t, U u)	two	boolean	Use when Predicate needs two input parameters
BiFunction<T, U, R>	R apply(T t, U u)	two	Any type	Use when Function needs two input parameters
BiConsumer<T, U>	void accept(T t, U u)	two	Nothing	Use when Consumer needs two input parameters
UnaryOperator<T>	public T apply(T t)	one	Any Type	Use this when input type & return type are same instead of Function<T, R>
BinaryOperator<T>	public T apply(T t, T t)	two	Any Type	Use this when both input types & return type are same instead of BiFunction<T, U, R>

96. Stream API in Java
    Refer to this cheat sheet for a succint overview of Stream API.

97. What are the methods available in Stream API?
    Refer to this image for a list of headers of all the methods in Stream API.

98. What are some operations that we can do using Java streams?
    Refer to this cheat sheet.

99. What is short-circuiting in Java streams?
    Short-circuiting terminal operations are some kind of operations where we can "short-circuit" the stream, i.e. interrupt the processing of the stream as soon

as we've found what we were looking for. For example if we are processing a `noneMatch` terminal operation, we'll finish the processing as soon as one element matches the criteria.

Short-circuiting is used with certain operations like: `limit, findFirst, findAny, anyMatch, allMatch or noneMatch` where we don't need to process the whole collection to get to a final result.

100. Mention some new features added in **Java 8**

Some notable features are:

- Lambda expressions
- Method references
- Functional interfaces
- Stream API
- Default methods in interface
- Static methods in interface
- Optional class
- `Collectors` class
- `forEach()` method

Refer to this image for all the features added in Java 8.

101. Serialization, customized serialization
102. What are the key points for developing a clean Object Oriented program?
    a. There are several key points for developing a clean object-oriented code that must be kept in mind. They are:
    b. Program to an interface (or the super type) not the implementation.
    c. Interacting of Classes should be loosely coupled among themselves.
    d. Code should implement tightly encapsulation. Avoid the use of public, static variables, and singleton design pattern wherever possible.
    e. Always reuse the code using inheritance, composition, and utility methods.
    f. Has-A relationship is better than Is-A relationship because it provides more flexibility.
    g. In the case of multi-threading applications, use immutable objects to represent the state.
    h. Make proper use of design patterns wherever possible.
    i. Use up-to-date software dependencies and make the best use of the latest technology available to us.

General HR

1. Do u have any questions for me
2. What is special about your city
3. Explain how you are a team leader