SE INTERVIEW QUESTIONS
https://www.edureka.co/blog/interview-questions/java-interview-questions/

FROM RAYTHEON
https://blog.udemy.com/java-interview-questions/

Behavioral Interview
https://www.theladders.com/behavioral-interview-questions/software-development

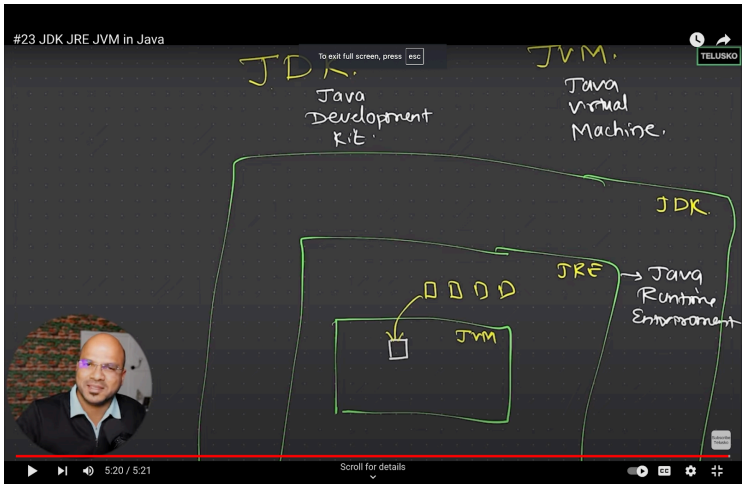
What is Java?
Java is a platform-independent high-level programming language. It is platform-independent because its byte codes can run on any system regardless of its operating system.

Java is a multi-platform, object-oriented, and network-centric language that can be used as a platform in itself. It is a fast, secure, reliable programming language for coding everything from mobile apps and enterprise software to big data applications and server-side technologies.
Used for:
https://aws.amazon.com/what-is/java/

What are the features of Java?
- Object-oriented programming (OOP) concepts
- Platform independent
- High performance (built in compiler) - JIT
- Multi-threaded

High Performance
Java offers high performance using the **JIT (Just In Time) compiler**. The compiler *only compiles that method which is being called*.
-The JIT enhances the performance of **interpreting byte code** by caching interpretations.
https://www.shiksha.com/online-courses/articles/features-of-java-programming-language/#:~:text=High%20Performance,byte%20code%20by%20caching%20interpretations.

JDK compiles code, **JVM** space where execute and runs code, **JRE** - jvm a part of JRe. Jvm needs environment where all classes and files are
https://www.youtube.com/watch?v=s7UgQ7_1KQY

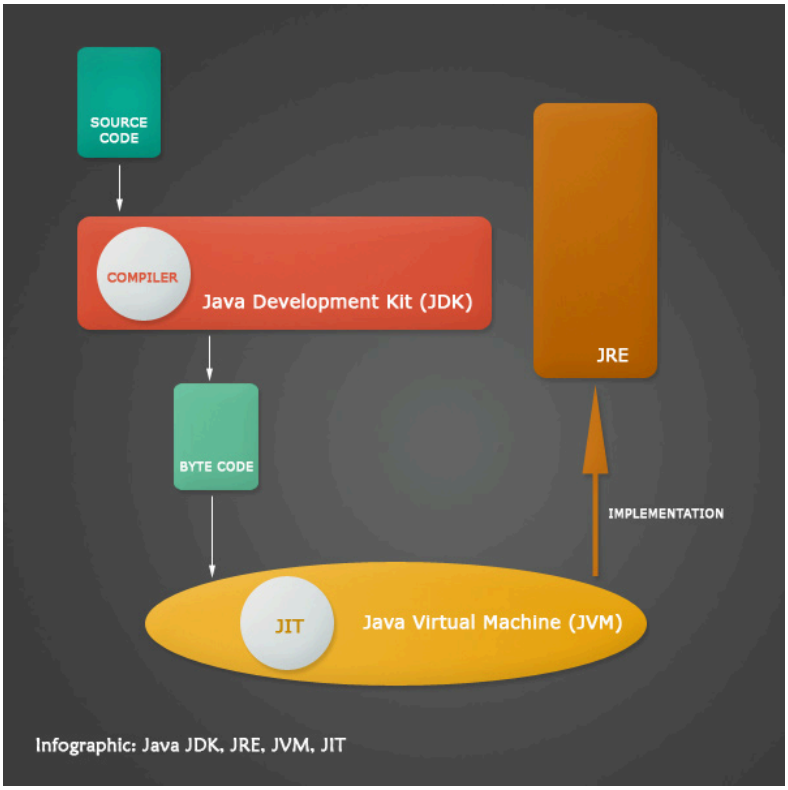


What is a JIT compiler?
JIT compiler is a component that is part of the JVM and hence its implementation JRE.
-JIT runs when the Java program is interpreted and its objective is to optimize the Java program and make it efficient in terms of performance.
https://www.tutorialwalk.com/java/what-is-jvm-jre-jdk-and-jit.html

How does a JIT compiler differ from a standard compiler?
JIT can access dynamic runtime information, and a standard compiler does not. Therefore, JIT can better optimize frequently used inlining functions.

https://www.tutorialwalk.com/java/what-is-jvm-jre-jdk-and-jit.html

- **Java Virtual Machine (JVM)** is an abstract definition of a computing machine.
- **Java Runtime Environment (JRE)** is the implementation of the JVM.
- **Java Development Kit (JDK)** is a set of tools using which Java programs can be developed and compiled.
- **Just In Time Compiler (JIT)** runs on the fly in the execution environment to make optimizations on the program.



"How would you code a function which returns the nth term of the Fibonacci series?"
"Define 'Encapsulation'."
"Define 'Polymorphism.'"
"Define 'Inheritance'."
"Define 'Arrays'."
"Define 'Linked lists'."
"Define 'Trees'."
SQL queries
derived classes(sub class), iterating through objects etc

Why is Main Static?
The main() method is declared static so that JVM can call it without creating an instance of the class containing the main() method. We must declare the main() function static as no class object is present when the java runtime starts. JVM can then load the class into the main memory and invoke the main() method.
https://www.scaler.com/topics/why-main-method-is-static-in-java/

What are the OOP concepts?

- Inheritance
- Encapsulation
- Polymorphism
- Abstraction
- Interface

Java - can implement multiple interfaces but only extend/inherit one class
https://stackoverflow.com/questions/21263607/can-a-normal-class-implement-multiple-interfaces
https://stackoverflow.com/questions/5836662/extending-from-two-classes

What is JDK?

- JDK stands for Java development kit.
- It can **compile**, document, and package Java programs.
- It contains both JRE and development tools.

What is JVM?

- JVM stands for Java virtual machine. ("write once, run anywhere")
- It is an abstract machine that provides a **run-time environment that allows programmers to execute Java bytecode.**
- JVM follows specification, implementation, and runtime instance notations.

What is JRE?

- JRE stands for Java runtime environment.
- JRE refers to a runtime environment that allows programmers to execute Java bytecode.
- **JRE is a physical implementation of the JVM.**

JVM vs JRE vs JDK - https://www.ibm.com/think/topics/jvm-vs-jre-vs-jdk

Java is pass by value.
-The terms "pass-by-value" and "pass-by-reference" are talking about variables. Pass-by-value means that the value of a variable is passed to a function/method. **Pass-by-reference means that a reference to that variable is passed to the function.** The latter gives the function a way to change the contents of the variable.
-By those definitions, Java is always pass-by-value. Unfortunately, when we deal with variables holding objects we are really dealing with object-handles called references which are passed-by-value as well. This terminology and semantics easily confuse many beginners.
SEE CODE - https://stackoverflow.com/questions/40480/is-java-pass-by-reference-or-pass-by-value

THE OBJECT REFERENCES ARE PASSED BY VALUE - Objects are not passed by reference

Pass by reference

-Memory location is passed into a function and any change made affects that memory location directly.

Pass by value

-a copy local to the function is stored and modifications made will not affect the original value

Java is a high-level programming language

-Java supports primitive data types - byte, boolean, char, short, int, float, long, and double and hence it is not a pure object oriented language.

-**Instance variables** are those variables that are **accessible by all the methods in the class**
-Instance variable - a variable which is bounded to its object itself.
-An instance variable is defined inside the class and outside the method. The scope of the variables exists throughout the class.

-**Local variables** are those variables present **within a block**, function, or constructor and can be accessed only inside them.

== checks if both objects point to the same memory location whereas
.equals() evaluates to the comparison of values in the objects.

https://www.interviewbit.com/java-interview-questions/

When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object

In **Java**, a **static** member is a member of a class that isn't associated with an instance of a class. Instead, **the member belongs to the class itself.**

STATIC - https://www.geeksforgeeks.org/static-keyword-java/

-**static** - mainly for memory management
-The static keyword belongs to the class than an instance of the class
-The static keyword is used for a constant variable or a method that is the same for every instance of a class
STATIC - When a member (block, variable, method, nested class) is declared static, it can be accessed before any objects of its class are created, and without reference to any object

-**Static Classes** - Static classes are used as inner classes so you **do not have to initiate the inner class**.

```
Class OuterClass {
        Static String message =”Suck it”;

        Static class NestedStaticClass {
                Public void printMessage() {
                        System.out.println(message);
                }
        }
        Public class InnerClass {
                Public void display() {
                        System.out.println(message);
                }
        }
}
Class Main {
        Public static void tacos() {
                System.out.println(“tacos”);
        }
        public static void main(String args[]) {

                // create instance of nested Static class
                 OuterClass.NestedStaticClass printer = new OuterClass.NestedStaticClass();
                printer.printMessage();
                 tacos();
```

```
        // In order to create instance of Inner class we need an Outer class
        // instance. Create Outer class instance for creating non-static nested class
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner  = outer.new InnerClass();
        inner.display();
    }
}
```

-**Static methods** are the methods in Java that can be called without creating an object of class

-**Static variable** - When a variable is declared as static, then a single copy of variable is created and shared among all objects at class level.

You cannot use the static keyword with a class unless it is an inner class.

A **static inner class** is a nested class which is a static member of the outer class.

It can be accessed without instantiating the outer class, using other static members.

Just like static members, a static nested class does not have access to the instance variables and methods of the outer class.

-what are static variables, classes, and methods used for? Why use them/when to use them? If you can pass static variables to other functions and classes, how would you do that?

Exception vs Errors - https://www.geeksforgeeks.org/errors-v-s-exceptions-in-java/

Try Catch Finally - https://www.geeksforgeeks.org/flow-control-in-try-catch-finally-in-java/

What are the differences between unchecked exception, checked exception, and errors?

- An **Unchecked exception** inherits from RuntimeException (which extends from exception). The JVM treats RuntimeException differently as there is no requirement for the application code to deal with them explicitly.
- A **checked exception** inherits from the exception class. The client code has to handle the checked exceptions either in a try-catch clause or has to be thrown for the super-class to catch the same. A checked exception thrown by a lower class (sub-class) enforces a contract on the invoking class (super-class) to catch or throw it.
- **Errors** (members of the error family) usually appear for more serious problems, such as OutOfMemoryError (OOM), that may not be so easy to handle.

Error : An Error "indicates serious problems that a reasonable application **should not try to catch.**

Errors are the conditions which cannot get recovered by any handling techniques. It surely cause termination of the program abnormally.

Errors are usually caused by serious problems that cannot be recovered from, while exceptions are used to handle recoverable errors within a program.

In java, both Errors and Exceptions are the subclasses of java.lang.

https://www.geeksforgeeks.org/errors-v-s-exceptions-in-java/

Handle exceptions with try catch

Error: THREE TYPES - **Syntax Error. Runtime Error. Logical Error.**

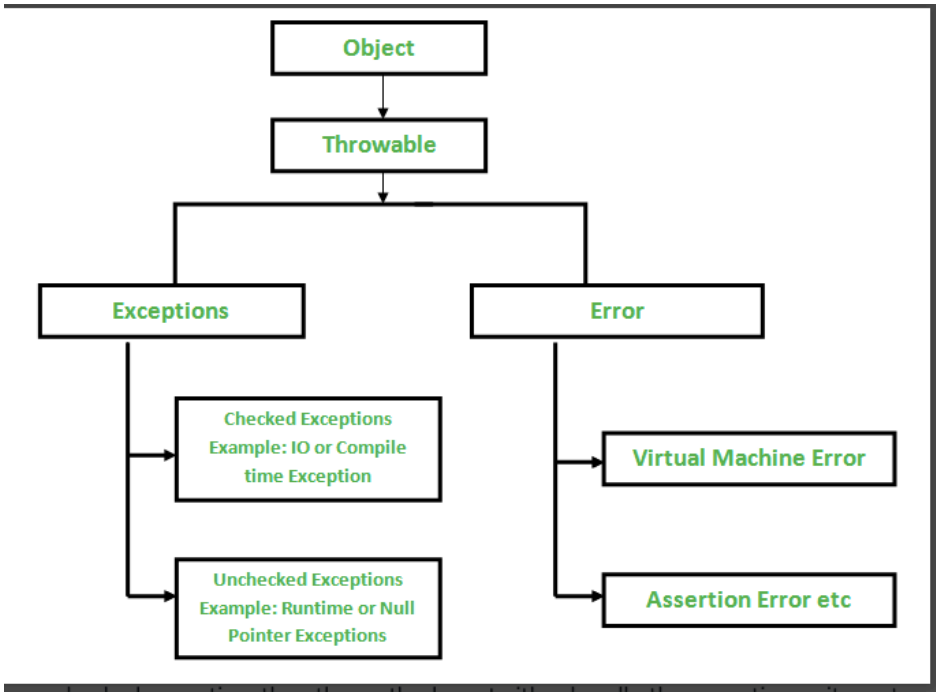
Error Examples:

- **OutOfMemoryError**: Thrown when the Java Virtual Machine (JVM) runs out of memory.

- **StackOverflowError**: Thrown when the call stack overflows due to too many method invocations (calls).
- **NoClassDefFoundError**: Thrown when a required class cannot be found.

Since **errors** are generally caused by problems that cannot be recovered from, **it's usually not appropriate for a program to catch errors**. Instead, the best course of action is **usually to log the error and exit the program.**

Checked Exception - checked at **compile time** - If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using the *throws* keyword.
-A **fully checked exception** is a checked exception where **all its child classes are also checked**, like IOException, and InterruptedException.
-A **partially checked** exception is a checked exception where some of its child classes are unchecked, like an Exception.
UNCHECKED Exception - In Java, exceptions under *Error* and *RuntimeException* classes are unchecked exceptions, everything else under throwable is checked. **runtime exceptions**



UNCHECKED EXAMPLES:

-Arithmetic Exception
-ArrayIndexOutOfBounds Exception

Ex: Runtime can be a reason for error:

```
public static void test(int i) {
        if (i == 0)
            return;
        else {
            test(i++);
        }
}
```

```
Exception in thread "main" java.lang.StackOverflowError
    at StackOverflow.test(ErrorEg.java:7)
```

An **Exception** "indicates conditions that a reasonable **application might want to catch."**

Exceptions are the conditions that occur at runtime and **may cause the termination of program. But they are recoverable using try, catch and throw keywords.**

```
int a = 5, b = 0;
// Attempting to divide by zero
try {
   int c = a / b;
}
catch (ArithmeticException e) {
   e.printStackTrace();
}
System.out.println("Tacos --");
```

This will print:

```
java.lang.ArithmeticException: / by zero
        at Main.main(Main.java:62)
Tacos --
// it will not terminate the program
```

Try Catch FINALLY - The finally block *always* executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break. Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.

-When to use error handling exceptions. Why use them? What happens to the program if you do not use error handling? What happens to your program if you don't use error handling for REST or importing information/data?

Final - https://www.javatpoint.com/final-keyword

The **final keyword** is for variable, method and class

Final Variable - If you make any variable as final, you cannot change the value of the final variable(It will be constant).

If you change the value - Output:Compile Time Error

Final Method - If you make any method as final, you cannot override it.

```
class Bike{
  final void run(){System.out.println("running");}
}
class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}
```

```
  public static void main(String args[]){
  Honda honda= new Honda();
  honda.run();
  }
}
```

Output:Compile Time Error

Final Class - If you make any class as final, you cannot extend it.

```
final class Bike{}
 class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda();
  honda.run();
  }
 }
```

Override - https://www.geeksforgeeks.org/overriding-in-java/

-**Override method** - When a method in a subclass has the **same name, same parameters and same return type(**or sub-type) **as a method in its super-class**, then the method in the subclass is said to override the method in the super-class. Use @Override above the subclass method when overriding
Declare the parent method as final if you do not want it to be overriden. Ex:

```
Class ParentClass {
        Final void show() {   System.out.println(“Overriding is not allowed”);   }
        // if we want child to override it:
        // void show() {        System.out.println(“Overriding is allowed”);   }
}
Class ChildClass extends ParentClass {
        Void show() {   System.out.println(“Child class cannot override this method”);  }
        // @Override
        // Void show() {   System.out.println(“Child class override this method”);  }
}
```

Prints out:

```
13: error: show() in Child cannot override show() in Parent
   void show() {  }
        ^
  overridden method is final
```

If we call super.show(); in child overriden child method, the parent method will be shown. Ex:

```
Class ChildClass extends ParentClass {
        @Override
        Void show() {
                super.show();
```

```
                System.out.println("Child class can override this method");
        }
}
Class Main {
        Public static void main(String[] args) {
                Parent obj = new Child();
                obj.show();
        }
}
```

method signature - is part of the **method** declaration. It's the combination of the **method name and the parameter list**

Overload method - allows different methods to have the same name, but different signatures where the signature can **differ by the number of input parameters** or **type of input parameters** or **both**. **Overloading is related to compile-time (or static) polymorphism.**

-Four principles of Object Oriented Programming -
(Encapsulation, Abstraction, Inheritance, and Polymorphism)

Encapsulation - Encapsulation is a concept in Object-Oriented Programming for combining properties and methods in a single unit.
-Encapsulation helps developers **follow a modular approach** for software development because each object has its **own set of methods and variables and serves its functions independent of other objects.**
-In addition to that, encapsulation serves **data hiding purposes.**
https://blog.udemy.com/java-interview-questions/

Encapsulation - the process by which data (variables) and the code that acts upon them (methods) are integrated as a single unit. **By encapsulating a class's variables, other classes cannot access them, and only the methods of the class can access them.**
https://www.simplilearn.com/tutorials/java-tutorial/java-encapsulation#:~:text=Encapsulation%20in%20Java%20is%20the,the%20class%20can%20access%20them.

Encapsulation - Encapsulation helps with data security, allowing you to **protect the data stored in a class** from system-wide access. As the name suggests, it safeguards the internal contents of a class like a capsule.
https://raygun.com/blog/oop-concepts-java/#encapsulation

ABSTRACTION - Abstraction aims to hide complexity from users and show them only relevant information.
https://raygun.com/blog/oop-concepts-java/#abstraction

-**Abstraction** - **Hiding** the **internal implementation** of the feature and **only showing the functionality** to the users. i.e. what it works (showing), how it works (hiding). Both abstract class and interface are used for abstraction.

Abstraction - Abstract means a concept or an Idea which is not associated with any particular instance
Using abstract class/Interface we express the intent of the class rather than the actual implementation
https://www.geeksforgeeks.org/difference-between-abstract-class-and-interface-in-java/
Applying abstraction means that each object should **only** expose a high-level mechanism for using it.

Interface vs Abstract Class

- The biggest difference between abstract and non abstract method is that
-abstract methods can either be hidden or overridden, but
-non abstract methods can only be hidden.
-And that abstract methods don't have an implementation, not even an empty pair of curly braces.
https://stackoverflow.com/questions/35738565/difference-between-methods-of-abstract-and-methods-of-non-abstract-in-c#:~:text=Conclusion%3A%20The%20biggest%20difference%20between,empty%20pair%20of%20curly%20braces.
VIDEO - https://www.youtube.com/watch?v=2aQ9Y7bumts

Difference between abstract class and interface:-
Definition:
-An abstract class is a class that cannot be instantiated and **can contain both abstract and non-abstract methods**.
-An interface, on the other hand, is a contract that specifies a set of methods that a class must implement.
Method implementation:
-In an abstract class, some methods can be implemented, while others are left abstract, meaning that they have no implementation and must be overridden by concrete subclasses.
-In contrast, all methods in an interface are by default abstract and must be implemented by any class that implements the interface.
https://www.geeksforgeeks.org/difference-between-abstract-class-and-interface-in-java/

Methods - Interface can have only abstract methods. methods contain **no body**. use interface if you want to use these methods, but how the methods work is different
Methods - Abstract class can have abstract and non-abstract methods.
Variables - **Interface** - In Java**,** are by default final.
Variables - **Abstract class** may contain non-final variables.
USE
Interface - It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.
Interface - Interfaces specify **what a class must** do/have and not how. It is the blueprint of the class.
Abstraction is a process of hiding the implementation details and showing only how or what it works (showing), how it works (hiding) - methods have a body.
Abstract classes are similar to interfaces.
Abstract - You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation.
-However, with abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods.
-With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default methods) are public. In addition, you can extend only one class, whether or not it is abstract, whereas you can implement any number of interfaces.
https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

Abstract class Shape {
Abstract void area() { } }
Square extends Shape {

@override
area() { } }

```
public interface MusicList {
        public int getNumChannels();
        public float getSampleRate();
        public int getNumSamples();
}
public class MusicLinkedList implements MusicList {
        @Override
        public int getNumChannels() { }
}
```

Inheritance - parent & child classes. Code reusability. Extend class

polymorphism - gives a way to use a class exactly like its parent so there's no confusion with mixing types. **But each child class keeps its own methods as they are.**

This typically happens by defining a (parent) interface to be reused. It outlines a bunch of common methods. Then, each child class implements its own version of these methods.

Polymorphism - It means one name, many forms. It is further of **two types — static and dynamic.**

Polymorphism - **OVERLOAD AND OVERRIDE**

Static polymorphism is achieved using **method overloading**

-and **dynamic polymorphism** using **method overriding**. It is closely related to inheritance. We can write a code that works on the superclass, and it will work with any subclass type as well.

Polymorphism - being able to assign a different meaning or usage to something in different contexts.

-To allow an entity such as a variable, a function, or an object to have more than one form.

-polymorphism gives a way to use a class exactly like its parent so there's no confusion with mixing types. But each child class keeps its own methods as they are.

-Polymorphism is one interface with many implementations

Data Structures/Collections/Containers

Lists

Queues

Set

Map

Hashmap vs HashTable

Answer: The difference between HashMap and HashTable can be seen below:

HashMap	HashTable
Methods are not synchronized	Key methods are synchronized
Not thread safety	Thread safety
Iterator is used to iterate the values	Enumerator is used to iterate the values
Allows one null key and multiple null values	Doesn't allow anything that is null
Performance is high than HashTable	Performance is slow

Hashing in Java is a technique for mapping data to a secret key that can be used as a unique identifier for data.

-What is a hashmap and when do you use it? What happens on the Java end of a hashmap/Explain hashing?
-Name some JUnit Annotations. When do you use them and why?
-Talk about how you connected your Spring application to your SQL database. What did you use and why?
-If given two strings, how would you return only the unique chars in N+N
-Git commands - push to master
-Difference between linked list and array. Iteration, access.

Java Primitive Data Types:

- **byte**: The byte data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive)
- **short**: The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive).
- **int**: By default, the int data type is a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of $2^{31}-1$.
- **long**: The long data type is a 64-bit two's complement integer. The signed long has a minimum value of -2^{63} and a maximum value of $2^{63}-1$.
- **float**: The float data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language
- **double**: The double data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification.
- **boolean**: The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.
- **char**: The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).

Dynamic: Java is a dynamic programming language. It supports dynamic loading of classes which means **classes are loaded on demand.**

Java is pass by value

Final Class keyword is used with the class to make sure that any **other class can't extend it.**
Final Method - cannot be overriden by a subclass.

Access Modifiers: Access Modifiers are the keywords which are used for set accessibility to classes, methods, and other members. In Java, these are the four access modifiers:
Public: can be accessed by any class or method.
Protected: The access level of a protected modifier is **within the package and outside the package through child class.**
Default: Default is accessible **within the package only**. All the classes, methods, and variables are of default when the public, protected, or private are not used.
Private: Classes, methods, and variables which are defined as private can be accessed within the class only.

Enum - field consists of fixed sets of constants

What is the difference between equals() and == in Java?
== checks if both objects **point to the same memory location** whereas
.equals() evaluates to the comparison of values in the objects.
https://www.geeksforgeeks.org/difference-equals-method-java/

An object has: state, behavior, and identity.

Q18. What are the main concepts of OOPs in Java?

1. **Inheritance**: Inheritance is a process where one class acquires the properties of another.
2. **Encapsulation**: a mechanism of wrapping up the data and code together as a single unit.
3. **Abstraction**: hiding the implementation details from the user and only providing the functionality to the users.
4. **Polymorphism**: the ability of a variable, function or object to take multiple forms.

Stack Vs Heap
Stack memory only contains local primitive and reference variables to objects in heap space.
Heap - Whenever an object is created, it's always stored in the Heap space.

In Java, what are the differences between heap and stack memory?
https://blog.udemy.com/java-interview-questions/
Memory
- Stack memory is used only by one thread of execution.
- All the parts of the application use heap memory.

Access
- Other threads can't access stack memory.
- Objects stored in the heap are globally accessible.

Memory Management
- Stack follows the LIFO manner to free memory.
- Memory management for heap stems from the generation associated with each object.

Lifetime
- Stack exists until the end of the execution of the thread.
- Heap memory lives from the start till the end of application execution.

Usage
- Stack memory only contains local primitive and reference variables to objects in heap space.
- Whenever you create an object, it is always stored away in the heap space.

this() - the current instance of a class
super() - the current instance of a parent/base class

Strings are immutable

Array vs ArrayList
ArrayList - Can contain values of different data types

Arrays - Need to specify the index in order to add data

Constructors - block of code which is used to initialize an object.

Method Overriding - to “Change” existing behavior of the method.

Servlets receive and respond to requests from Web clients, usually across HTTP, the HyperText Transfer Protocol.

Throw vs Throws - https://beginnersbook.com/2013/04/difference-between-throw-and-throws-in-java/
Throws clause is used to **declare an exception**, which means it works **similar to the try-catch block**.
On the other hand, throw keyword is used to **throw an exception** explicitly.
If we see syntax wise than **throw** is followed by an instance of Exception class and **throws** is followed by exception class names.

Throw vs throws

Throws can be multiple exceptions, throw is singular. Also throws uses matching to throw the right one.

For example:
throw new ArithmeticException("Arithmetic Exception");
And
throws ArithmeticException;

Throw keyword is used in the method body to **throw an exceptio**n,
while **throws** is used in method signature to **declare the exceptions** that can occur in the statements present in the method.

```
void myMethod() {
  try {
    //throwing arithmetic exception using throw
    throw new ArithmeticException("Something went wrong!!");
  }
  catch (Exception exp) {
    System.out.println("Error: "+exp.getMessage());
  }
}
//Declaring arithmetic exception using throws
void sample() throws ArithmeticException{
  //Statements
}
```

Java 8 Interview Questions - USE THIS: https://www.tutorialspoint.com/java8/java8_method_references.htm

```
/* Lambda expressions - is a short block of code which takes in parameters and returns a value.
Can express instances of functional interfaces
(An interface with single abstract method is called functional interface.
An example is java.lang.Runnable). lambda expressions implement the only
abstract function and therefore implement functional interfaces */

/* A functional interface is an interface that contains only one abstract method. */

/*  Method reference is used to refer method of functional interface */

/* Optional is a container object used to contain not-null objects.
Optional object is used to represent null with absent value.
This class has various utility methods to facilitate code to handle
values as 'available' or 'not available' instead of checking null values.
https://www.tutorialspoint.com/java8/java8_optional_class.htm */

/* DEFAULT METHODS - Before Java 8, interfaces could have only abstract methods.
The implementation of these methods has to be provided in a separate class.
So, if a new method is to be added in an interface, then its
implementation code has to be provided in the class implementing
the same interface.
To overcome this issue, Java 8 has introduced the concept of default methods
which allow the interfaces to have methods with implementation without
affecting the classes that implement the interface.
**Java 8 - 'List' or 'Collection' interfaces do not have 'forEach' method declaration.
Thus, adding such method will simply 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 implement the same. */

/*Streams - JS type functions (map, filter, etc)
* process collections of objects. A stream is a sequence of objects
* that supports various methods which can be pipelined to produce the desired result. */

/* Spliterators, like other Iterators, are for traversing the elements of a source.  */
```

annotations - information for the compiler
@override,
can create your own annotations

```
 // Annotating return type of a function
   static @TypeAnnoDemo int abc(){
      System.out.println("This function's  return type is annotated");
      return 0;
   }
}
```

jUnit `@Test` - test case
`@before` - before test case
`@after` - after test case

Coupling - Coupling refers to the usage of an object **by another object.** It can also be termed as **collaboration**. how often do changes in class A force related changes in class B. Two types of Coupling: Tight coupling and Loose coupling

Tight coupling - Tight coupling means the two classes often change together.

```
// tight coupling
class Volume {
    public static void main(String args[]) {
        Box b = new Box(5,5,5);
        System.out.println(b.volume);   }   }
class Box {
    public int volume;
    Box(int length, int width, int height) {
        this.volume = length * width * height;  } }
```

Loose Coupling - loose coupling means they are mostly independent.

```
// loose coupling concept
public interface Topic {
   void understand();   }
class Topic1 implements Topic {
         public void understand() {   System.out.println("Got it");   } }
class Topic2 implements Topic {
         public void understand()  {  System.out.println("understand");  }   } public class Subject {
         public static void main(String[] args) {
       Topic t = new Topic1();
       t.understand();   }   }
```

https://www.geeksforgeeks.org/coupling-in-java/

property files -

check exception vs unchecked exception

Checked: are the exceptions that are **checked at compile time**. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.

Unchecked: Unchecked are the exceptions that are **not checked at compiled time**. In Java exceptions under *Error* and *RuntimeException* classes are unchecked exceptions, everything else under throwable is checked.

```
class Main {
  public static void main(String args[]) {
    int x = 0;  int y = 10;
    int z = y/x;  }   }
// this compiles fine, but will get an java.lang.ArithmeticException: error
```

HashMap vs HashTable

Answer: The difference between HashMap and HashTable can be seen below:

HashMap	HashTable
Methods are not synchronized	Key methods are synchronized
Not thread safety	Thread safety
Iterator is used to iterate the values	Enumerator is used to iterate the values
Allows one null key and multiple null values	Doesn't allow anything that is null
Performance is high than HashTable	Performance is slow

HashSet vs TreeSet

Answer: The difference between HashSet and TreeSet can be seen below:

HashSet	TreeSet
Inserted elements are in random order	Maintains the elements in the sorted order
Can able to store null objects	Couldn't store null objects
Performance is fast	Performance is slow

https://www.softwaretestinghelp.com/core-java-interview-questions/

Exception Propagation.

Exception is first thrown from the method which is **at the top of the stack.**

-If it doesn't catch, then it pops up the method and moves to the previous method and so on until they are got.

```
public class Manipulation{
public static void main(String[] args){
add();
}
public void add(){
addition();
}
```

ANSWER - addition(); add(); main();

MultiThreading

-Multithreading is the ability of a program or an operating system to enable more than one user at a time without requiring multiple copies of the program running on the computer.

-Multithreading is a programming concept in which the application can create a small unit of tasks to execute in parallel

https://www.geeksforgeeks.org/multithreading-in-java/

https://www.learnthepart.com/course/38e044d7-9a57-47a1-8f02-3e3ea31ea7ec/d3785a9e-01e7-483c-b594-5a8c47c71752

Differences Between Thread and Runnable:
https://naveen-metta.medium.com/understanding-the-difference-between-thread-and-runnable-in-java-dc869849b8c7

Extending vs. Implementing:
One significant distinction between Thread and Runnable arises from Java's support for single inheritance. **A class extending Thread cannot extend any other class**, limiting its flexibility. Conversely, a class can implement multiple interfaces, allowing it to implement Runnable and extend another class if necessary.

Reusability:
The concept of reusability is better facilitated by using Runnable. By separating the task from the thread, code becomes more modular, promoting better object-oriented design. With Thread, the task and the thread are tightly coupled in a single class, potentially limiting reusability.

```
class ReusableTask implements Runnable {
   public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Reusable Task: " + i);
        } }  }

public class ReusabilityExample {
   public static void main(String[] args) {
        ReusableTask reusableTask = new ReusableTask();

        Thread thread1 = new Thread(reusableTask);
        thread1.start();

        Thread thread2 = new Thread(reusableTask);
        thread2.start();     }    }
```

In this example, the ReusableTask is a Runnable implementation, and two threads (thread1 and thread2) share the same instance of the task, showcasing reusability.

Resource Sharing:
When multiple threads share resources, implementing **Runnable** is often the preferred approach. This is because a single instance of the task can be shared among multiple threads, promoting better resource sharing and reducing potential resource contention. With Thread, each thread has its instance, which might lead to increased memory consumption.

Thread Pooling:
Java provides the Executor framework for managing and controlling thread execution. When using Runnable, it becomes easier to work with thread pools. The **ExecutorService interface** and the **Executors** utility class facilitate the creation of thread pools and the execution of Runnable tasks.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```

```
class TaskInThreadPool implements Runnable {
   public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread " + Thread.currentThread().getId() + ": " + i);
        }  }  }



public class ThreadPoolExample {
   public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(2);


        for (int i = 0; i < 3; i++) {
            Runnable taskInPool = new TaskInThreadPool();
            executorService.execute(taskInPool);
        }
        executorService.shutdown();   }   }
```

In this example, a thread pool is created using the newFixedThreadPool method from the Executors class. The execute method is then used to submit Runnable tasks to the pool.

While Thread provides a straightforward mechanism for creating and managing threads, it comes with limitations due to Java's single inheritance constraint. On the other hand, Runnable promotes better object-oriented design, reusability, and resource sharing, making it a preferred choice in many scenarios.

CONCLUSION

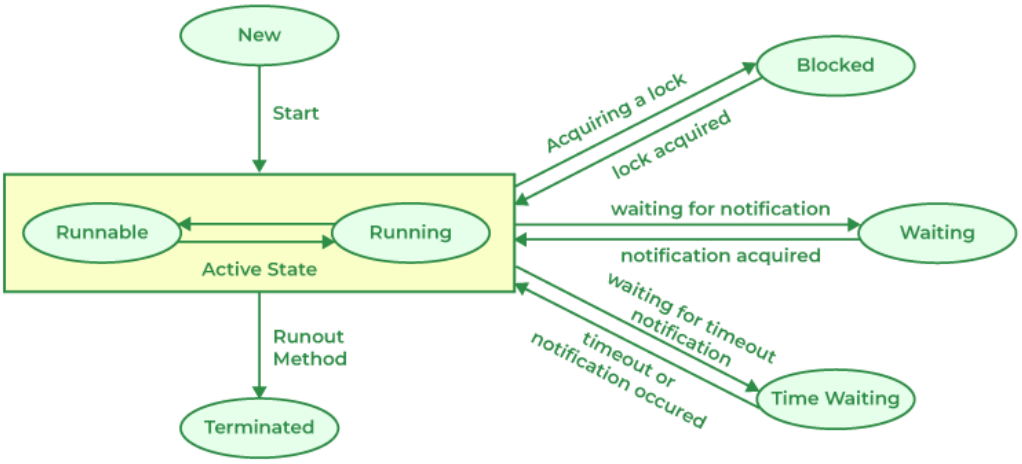
Create a separate class that implements the Runnable interface and create a Thread in a separate class without extending the Thread class.

https://naveen-metta.medium.com/understanding-the-difference-between-thread-and-runnable-in-java-dc869849b8c7

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. New State
2. Runnable State
3. Blocked State
4. Waiting State
5. Timed Waiting State
6. Terminated State

various states of a thread at any instant in time.



1. **New Thread**: When a new thread is created, it is in the new state. The thread **has not yet started to run** when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.
2. **Runnable State**: A thread that is **ready to run is moved to a runnable state**. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.
   A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.
3. **Blocked**: The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock.
4. **Waiting state**: The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will **notify or that** thread will be terminated.
5. **Timed Waiting**: A thread lies in a **timed waiting state** when it **calls a method with a time-out parameter**. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.
6. **Terminated State**: A thread terminates because of either of the following reasons:
   a. Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
   b. Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.

https://www.geeksforgeeks.org/lifecycle-and-states-of-a-thread-in-java/?ref=lbp

Deadlock vs Race Condition
A **race** condition occurs when two threads use the same variable at a given time.
Deadlock exists when two threads seek one lock simultaneously.
https://www.wallarm.com/what/what-is-a-race-condition

Deadlock - Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.
https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html

Deadlock - Synchronized keyword is the only reason for deadlock situation hence while using synchronized keyword we have to take special care.
Avoid Deadlock
-use **Thread.join()** method with the maximum time you want to wait for the thread to finish.

-Use **Lock Ordering**: We have to always assign a numeric value to each lock and before acquiring the lock with a higher numeric value we have to acquire the locks with a lower numeric value.
-**Avoiding unnecessary Locks**: We should use locks only for those members on which it is required, unnecessary use of locks leads to a deadlock situation. And it is recommended to use a lock-free data structure and If it is possible to keep your code free from locks. For example, instead of using synchronized ArrayList use the ConcurrentLinkedQueue.
-Solve - **lock hierarchy**

Race Condition - occurs when two or more threads can access shared data and they try to change it at the same time.
-Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data.
-Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.
SOLVE RACE CONDITION
To avoid race conditions, any operation on a shared resource – that is, on a resource that can be shared between threads – must be executed atomically.
-One way to achieve atomicity is by using critical sections — mutually exclusive parts of the program
-Solve with Lock & Unlock

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads.
https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html
Solve Starvation - use thread pools, which are collections of pre-created threads that can execute tasks from a queue.
https://www.linkedin.com/advice/3/how-can-you-prevent-thread-starvation-java-pnode#:~:text=One%20way%20to%20prevent%20thread,handle%20exceptions%20and%20shutdown%20gracefully.

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.
Threads can be created by using two mechanisms :

1. **Extending the Thread class**
2. **Implementing the Runnable Interface**

What is a thread?
-the flow of execution is called Thread.
Every java program has at least one thread called the main thread, the main thread is created by JVM.
The user can define their own threads by **extending the Thread class** (or)
-by **implementing the Runnable interfac**e. Threads are executed concurrently.

How do you make a thread in Java?
Answer: There are two ways available to make a thread.

a) Extend Thread class: Extending a Thread class and override the run method. The thread is available in java.lang.thread.

Example:

```
Public class Addition extends Thread {
   public void run () {  …  }
}
```

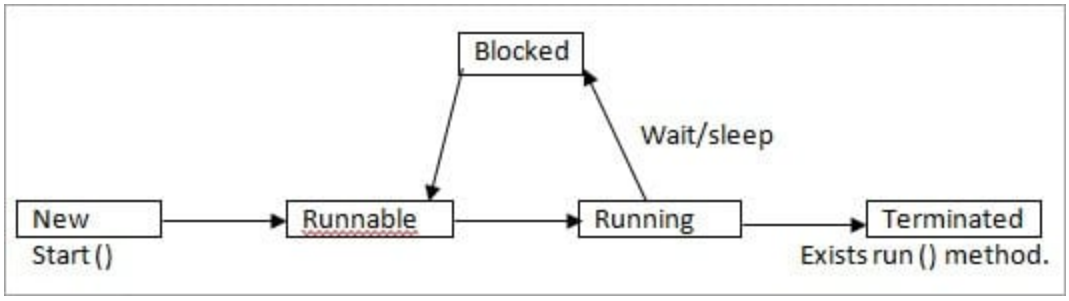
The disadvantage of using a thread class is that we cannot extend any other classes because we have already extended the thread class. We can overload the run () method in our class.

b) Implement Runnable interface: Another way is by implementing the runnable interface. For that, we should provide the implementation for the run () method which is defined in the interface.
Example:

```
Public class Addition implements Runnable {
   public void run () { … }
}
```

THREADS - https://www.softwaretestinghelp.com/core-java-interview-questions/

Explain the lifecycle of a thread
Thread has following states
-New
-Runnable
-Running
-Non-Runnable (Blocked)
-Terminated



- **New:** In New state, a Thread instance has been **created but start() method is not yet invoked**. Now the thread is not considered alive.
- **Runnable**: The Thread is in the runnable state after the invocation of the start() method, but before the run() method is invoked. But a thread can also return to the runnable state from waiting/sleeping. In this state, the thread is considered alive.
- **Running:** The thread is in a **running state after it calls the run()** method. Now the thread begins the execution.
- **Non-Runnable(Blocked)**: The thread is alive but it is not eligible to run. It is not in the runnable state but also, it will return to the runnable state after some time. Example: **wait, sleep, block.**
- **Terminated**: Once the run method is completed then it is terminated. Now the thread is not alive.

OOP vs Functional Programming
OOP focuses on objects, which are instances of a class, and their interactions with each other.
FP focuses on functions and their inputs and outputs.
OOP is based on the principles of encapsulation, inheritance, polymorphism, and abstraction.

https://fluxtech.me/blog/object-oriented-programming-vs-functional-programming/#:~:text=OOP%20focuses%20on%20objects%2C%20which,inheritance%2C%20polymorphism%2C%20and%20abstraction.

Advantages of OOP

One of the main advantages of OOP is that it enables code reuse and modularity. By defining classes, which are blueprints for objects, programmers can create multiple instances of the same type of object, and inherit common attributes and behaviors from parent classes.

Disadvantages

It can increase the complexity and size of the code, **especially when the system involves multiple levels of inheritance, polymorphism, and dynamic binding.**

-These features can make the code harder to understand, debug, and test, and can introduce errors and bugs that are difficult to detect and fix.

-Another drawback of OOP is that it can consume more memory and CPU resources than other paradigms, such as procedural or functional programming.

-This is because objects store both data and methods, and require more space and time to create and manipulate.

https://www.linkedin.com/advice/1/what-advantages-disadvantages-object-oriented-k0nlf

Disadvantages

The OOP paradigm has been criticized for overemphasizing the use of objects for software design and modeling at the expense of other important aspects (computation/algorithms).[29][30] For example, Rob Pike has said that OOP languages frequently shift the focus from data structures and algorithms to types. as opposed to functional programming

https://en.wikipedia.org/wiki/Object-oriented_programming

- Object-oriented languages are good when you have a fixed set of operations on things, and as your code evolves, you primarily add new things. This can be accomplished by adding new classes which implement existing methods, and the existing classes are left alone.
- Functional languages are good when you have a fixed set of things, and as your code evolves, you primarily add new operations on existing things. This can be accomplished by adding new functions which compute with existing data types, and the existing functions are left alone.

TDD - Test Driven Development

-write tests before development

-It follows a cyclical process of writing a failing test, writing the minimum code to make the test pass, and then refactoring the code.

https://www.browserstack.com/guide/what-is-test-driven-development

-Test-driven development (TDD) is a way of writing code that involves writing an automated unit-level test case that fails, then writing just enough code to make the test pass, then refactoring both the test code and the production code, then repeating with another new test case.

https://en.wikipedia.org/wiki/Test-driven_development