QC 1:

1)Architecture of Linux

- **Hardware**: The physical components of a computer system.
- **Kernel**: The core part of the Linux operating system that manages system resources and hardware communication.
- **Shell**: The interface that allows users to interact with the kernel.
- **Application**: Software that performs specific tasks for users.
- **Utility**: Essential tools and programs that perform a variety of tasks to maintain and configure the system.

2) Distributions

Distributions, or "distros," are different versions of the Linux operating system that include the Linux kernel, system utilities, applications, and package management systems. Each distribution is tailored to meet specific needs and preferences, offering various features, user interfaces, and software repositories.

Different Distributions in Linux

- **Ubuntu**: Known for its user-friendliness and widespread community support, ideal for both beginners and advanced users.
- **Fedora**: Sponsored by Red Hat, focuses on cutting-edge features and technologies.
- **Debian**: Known for its stability and extensive software repository.
- **Arch Linux**: A minimalist distribution that gives users complete control over their system.
- **CentOS**: A free, open-source version of Red Hat Enterprise Linux, known for its stability and performance in server environments.

- **Mint**: Based on Ubuntu, known for its ease of use and out-of-the-box functionality.
- **openSUSE**: Known for its powerful YaST configuration tool and strong community support.
- **Manjaro**: Based on Arch Linux, but more user-friendly with a focus on accessibility.

3) Directory Commands in Linux

- `ls`: Lists the contents of a directory.
- `cd`: Changes the current directory.
- `pwd`: Prints the current working directory.
- `mkdir`: Creates a new directory.
- `rmdir`: Removes an empty directory.
- `rm -r`: Removes a directory and its contents recursively.
- `mv`: Moves or renames files and directories.
- `cp -r`: Copies directories and their contents recursively.
- `find`: Searches for files and directories within a directory hierarchy.
- `du`: Estimates and displays the disk usage of files and directories.

4) File Handling Commands in Linux

- `touch`: Creates a new, empty file or updates the timestamp of an existing file.
- `cat`: Concatenates and displays the content of files.
- `cp`: Copies files from one location to another.
- `mv`: Moves or renames files.
- `rm`: Removes files.
- `head`: Displays the first few lines of a file.

- `tail`: Displays the last few lines of a file.
- `more`: Displays file content one screen at a time.
- `less`: Similar to `more`, but allows backward movement in the file.
- `chmod`: Changes the permissions of a file.
- `chown`: Changes the ownership of a file.
- `stat`: Displays detailed information about a file.
- `ln`: Creates hard and symbolic links to files.

5)Filter Commands in Linux

- `grep`: Searches for patterns in files and displays matching lines.
- `sed`: Stream editor for filtering and transforming text.
- `awk`: Pattern scanning and processing language.
- `sort`: Sorts lines of text files.
- `uniq`: Reports or omits repeated lines.
- `cut`: Removes sections from each line of files.
- `tr`: Translates or deletes characters.
- `wc`: Counts lines, words, and characters in files.
- `tee`: Reads from standard input and writes to standard output and files.
- `head`: Outputs the first part of files.
- `tail`: Outputs the last part of files.

--Scripting files are saving with .sh

6) What is MVC?

MVC stands for Model-View-Controller. It is a software design pattern commonly used for developing user interfaces that divides an application into three

interconnected components:

1. **Model**: Represents the data and the business logic of the application. It directly manages the data, logic, and rules of the application.
2. **View**: Represents the presentation layer. It displays the data provided by the model in a specific format.
3. **Controller**: Acts as an intermediary between the Model and the View. It listens to the input from the user, processes it (possibly altering the state of the Model), and updates the View accordingly.

This separation helps in organizing code, promotes reusability, and makes the application more manageable.'

7) What is Git?

Git is a distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

-It allows multiple developers to work on the same project simultaneously without interfering with each other's work.

-Git tracks changes in the source code, enabling developers to collaborate, manage revisions, and maintain a history of project development. It supports branching and merging, which makes it easy to experiment with new features and integrate them into the main project.

8) What is Staging Area?

-In the context of Git, a **Staging Area** (also known as the "index") is an intermediate space where changes are collected before they are committed to the repository.

-Temp space where files are stored before making an commit

-Git is so smart that it automatically detects the files that have been change and only those files are added in staging area.

-It is also called as adding file to index.

How the Staging Area Works:

1. **Making Changes**:

- When you modify files in your working directory, these changes are untracked or modified but not yet part of any commit.

2. **Staging Changes**:
   - You use the `git add` command to add specific changes to the Staging Area. This allows you to prepare only certain parts of your changes for the next commit.
   - For example, `git add file.txt` will stage the changes in `file.txt`.
3. **Committing Changes**:
   - Once you've staged the changes you want to include, you commit them with `git commit`. This moves the changes from the Staging Area to the commit history in the repository.

9) What is Branch?

-In Git, a **branch** is a parallel version of your code that diverges from the main working project.

-It allows you to work on different tasks, features, or experiments independently without affecting the main codebase until you're ready to merge those changes back.

-git checkout branch-name —switch branch

-git checkout -b branch-name —switch as well as create it

10) What are Merge Conflicts?

-Merge conflicts occur when Git is unable to automatically resolve differences between two branches or commits that are being merged.

-This usually happens when the same line of code or the same file has been edited in conflicting ways in the branches being merged.

- In such cases, Git requires manual intervention to resolve the conflicts before the merge can be completed.

Common Causes of Merge Conflicts:

1. **Conflicting Changes**: Two developers modify the same lines of code in different branches.
2. **File Deletion**: One branch deletes a file while another branch modifies it.
3. **File Rename**: A file is renamed in one branch but not in another, leading to confusion during the merge.

11) What is Pull Request?

A Pull Request (PR) is a feature in Git-based version control systems like GitHub, GitLab, and Bitbucket that It allows developers to notify team members that they have completed a feature, bug fix, or other work and would like it to be reviewed and potentially merged into the main branch of the project.

How a Pull Request Works:

1. **Create a Branch**:
   - Developers create a new branch to work on a specific feature or fix.
2. **Make Changes**:
   - They commit the changes to this branch.
3. **Open a Pull Request**:
   - Once the changes are ready, they open a Pull Request to merge the changes from their branch into the main branch or another target branch.
   - **CI/CD Integration**: Often, automated tests and other Continuous Integration/Continuous Deployment (CI/CD) processes are triggered when a PR is opened. This helps catch bugs or issues before the code is merged.
4. **Review and Discussion**:
   - Other team members review the changes, discuss potential issues, and provide feedback.
5. **Merge**:
   - After approval, the changes are merged into the target branch.

Pull Requests are crucial for collaboration, ensuring code quality, and maintaining a clear history of changes.

12) What is git ignore and why we need it?

`.gitignore` is a file in a Git repository that specifies which files and directories should be ignored by Git. This means that the files listed in `.gitignore` will not be tracked, staged, or committed to the repository.

Why We Need `.gitignore`:

1. **Prevent Unnecessary Files**: It helps prevent unnecessary files such as temporary files, build artifacts, and system-specific files from being added to the repository.
2. **Reduce Repository Size**: By ignoring large files that are not essential to the source code, `.gitignore` helps keep the repository size manageable.
3. **Maintain Clean History**: It ensures that the commit history remains clean and free from irrelevant files, making it easier to understand the changes and collaborate with others.
4. **Enhance Security**: Sensitive files like configuration files containing passwords or API keys can be excluded to enhance security.

Example of a `.gitignore` File:

```
# Ignore all .log files
*.log

# Ignore the node_modules directory
node_modules/

# Ignore compiled files
*.class

# Ignore system files
```

```
.DS_Store
Thumbs.db
```

13) Clone vs Fork

Clone:

- Cloning a repository means creating a local copy of that repository on your machine.
- The cloned repository has a direct connection to the original repository, allowing you to pull updates from it and push changes back to it (if you have the necessary permissions).
- Use the `git clone` command to clone a repository.

Fork:

- Forking a repository is creating a copy of the repository on your own GitHub account.
- The forked repository is independent of the original repository, but you can still pull updates from the original repository to your fork.
- Forking is often used when you want to contribute to a project but do not have direct write access to the original repository.
- After making changes in your forked repository, you can create a pull request to propose merging your changes back into the original repository.

When to Use Clone:

- When you have write access to the repository and need to contribute directly.
- When you want to keep your local repository synchronized with the original one.

When to Use Fork:

- When you do not have write access to the repository but want to contribute to it.
- When you want to make changes without affecting the original repository until your changes are reviewed and accepted.

In summary, cloning is for creating a local copy of a repository you intend to work on directly, while forking creates an independent copy on your account for proposing changes to the original repository.

1.

1. Three Input Streams:

- **Input Stream (`System.in`)**: This is the standard input stream, typically used to receive input from the user via the console.
- **Output Stream (`System.out`)**: This is the standard output stream, used to print output to the console.
- **Error Stream (`System.err`)**: This is the standard error stream, used to output error messages. It works similarly to `System.out`, but it's typically used for error handling.

2. `System.out.println()` and `out`:

- `System.out.println()`: This is a method that prints a message to the console followed by a newline.
- `out`: It is a static member of the `System` class and is an instance of `PrintStream`. The `out` object is connected to the console by default.

3. Overriding `public static void main(String[] args)`:

- **No, you cannot override `main`**: The `main` method is static, and static methods cannot be overridden. However, you can overload the `main` method by creating multiple versions with different parameter lists.

4. JRE, JVM, JDK:

- **JRE (Java Runtime Environment)**: It provides the libraries, Java Virtual Machine (JVM), and other components to run applications written in Java. It doesn't contain tools for development like compilers.
- **JVM (Java Virtual Machine)**: It's an abstract machine that enables your computer to run Java programs. It provides a platform-independent execution environment.

- **JDK (Java Development Kit)**: It's a full-featured software development kit for Java, which includes the JRE and tools to develop, debug, and monitor Java applications.

5. Is Java Really Platform Independent?

- **Yes, Java is platform-independent**: Java programs are compiled into bytecode, which can be executed on any platform that has a JVM. The JVM abstracts the underlying OS, making Java programs platform-independent. While different JDKs are available for different platforms, they all produce platform-independent bytecode.

6. Wrapper Classes & Boxing:

- **Wrapper Classes**: These are classes that encapsulate a primitive data type into an object (e.g., `Integer` for `int`, `Double` for `double`). They provide utility methods to manipulate the values.
- **Boxing**: This is the process of converting a primitive type into a corresponding wrapper class object (e.g., converting `int` to `Integer`).

7. Automatic Boxing:

- **Automatic/Behind the Scene**: In Java, boxing is done automatically when you assign a primitive value to a variable of the corresponding wrapper class (e.g., `Integer i = 10;`).

8. Calling the Garbage Collector:

- **System.gc()**: This method suggests that the JVM performs garbage collection. However, it's just a request, and the JVM decides whether to actually perform garbage collection or not.

9. Immutability:

- **String**: Immutable because once a `String` object is created, it cannot be changed. Every modification creates a new `String` object.
- **StringBuffer**: Mutable and thread-safe. It can be modified without creating new objects, and is synchronized, which means it is safe to use in multi-threaded environments.

- **StringBuilder**: Similar to `StringBuffer`, but it is not synchronized, making it faster but unsafe for use in multi-threaded environments.

10. Difference between `StringBuffer` and `StringBuilder`:

- **StringBuffer**: Thread-safe and synchronized, but slightly slower.
- **StringBuilder**: Not thread-safe, not synchronized, but faster.

11. Synchronization Example - Traffic Signal:

- **Synchronization**: In Java, synchronization ensures that only one thread can access a resource at a time. A traffic signal is a good analogy—only one side is allowed to pass at a time, preventing accidents.

12. `public static void main(String ...args)`:

- **Varargs**: This syntax is correct and allows the main method to accept a variable number of arguments. It's functionally equivalent to `String[] args`.

Use of `super` Keyword:

- **Super**: This keyword is used to refer to the immediate parent class object. It's used to call the parent class's methods and constructors.
- -method
- -variable
- -constructors

15. What is Java? Why Java?

- **Java**: Java is a high-level, object-oriented programming language that is platform-independent due to its bytecode compilation. Java is known for its portability, security features, and robustness, making it ideal for enterprise-level applications.

16. What is an Object/Class?

- **Class**: A blueprint or template from which objects are created. It defines properties and behaviors (fields and methods).

- **Object**: An instance of a class containing actual values for the fields and can perform actions defined by methods.

17. Compiling and Executing a Java Program:

- **Compile**: Use `javac MyProgram.java` to compile the code into bytecode.
- **Execute**: Use `java MyProgram` to run the compiled bytecode on the JVM.

18. Main Method Signature:

- **Signature**: `public static void main(String[] args)`
- **String Array Population**: When a Java application is started, the JVM passes command-line arguments into the `main` method through the `args` array.

19. Root Class - `Object`:

- **Object Class**: The root class from which every Java class implicitly extends.

20. Methods Available in `Object` Class:

- **Key Methods**: `equals()`, `hashCode()`, `toString()`, `clone()`, `getClass()`, `notify()`, `notifyAll()`, `wait()`.

21. Primitive Data Types in Java:

- **Types**: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`.

22. Default Values for Data Types:

- **Default Values**: `int` - 0, `boolean` - false, `char` - '\u0000', `object references` - null.

23. Annotations:

- **Annotations**: Metadata that provides data about a program but is not part of the program itself. Examples include `@Override`, `@Deprecated`, and `@SuppressWarnings`.

24. Cloning an Object:

- **Cloning**: Use `clone()` method from `Object` class, which requires implementing `Cloneable` interface and overriding `clone()`.
- Copy constructor

- clone method using cloneable interface

25. Where are Strings Stored?

- **String Pool**: Strings are stored in a special memory area known as the String pool, which is part of the heap. When a new string is created, Java checks the pool to see if it already exists.

27. Forcing Garbage Collection:

- **Force GC**: While you can request garbage collection using `System.gc()`, you cannot force it. The JVM decides when to run the garbage collector.

28. Difference between `==` and `.equals()`:

- `==`: Compares memory addresses (reference comparison).
- `.equals()`: Compares the content of objects (value comparison).

29. Why are Strings Immutable in Java?

- **Immutability**: Strings are immutable to improve security, synchronization, and performance. Once created, a `String` object cannot be changed, ensuring that it's safe to share between multiple threads.
- -memory efficiency
- -security

30. Non-access Modifiers in Java:

- **Examples**: `static`, `final`, `abstract`, `synchronized`, `volatile`, `transient`.

31. Difference between `static` and `final` Variables:

- **Static**: Shared among all instances of a class.
- **Final**: Cannot be reassigned once assigned a value.

32. Data Types in `switch` Statements:

- **Supported Types**: `byte`, `short`, `int`, `char`, `String`, `enum`, and `Character`, `Byte`, `Short`, `Integer`.

33. Passing Multiple Values with a Single Parameter:

- **Varargs**: Use `...` (e.g., `

26. What is a POJO? What is a Bean?

POJO (Plain Old Java Object):

- A POJO is a simple Java object that does not adhere(stick to) to any special Java model or framework requirements.
- It typically has private fields with public getter and setter methods, but it doesn't necessarily follow any strict conventions.
- POJOs are used to increase the readability and reusability of a program.

Bean:

- A JavaBean is a specific type of POJO that follows certain conventions:
  - It must have a public default constructor.
  - It should provide public getter and setter methods to access its properties.
  - It should be serializable, which means it can be converted into a byte stream and subsequently restored.
- JavaBeans are used in various Java frameworks and are often employed in UI components, data transfer objects, and other areas where a standard structure is beneficial.

Use of `super` Keyword:

The `super` keyword in Java is used to refer to the immediate parent class object. It is commonly used in the following scenarios:

1. **Access Parent Class Methods**:
   - You can use `super` to call a method defined in the parent class.

```
class Parent {
    void display() {
        System.out.println("Parent class method");
    }
}

class Child extends Parent {
    void display() {
        super.display(); // Calls the parent class method
        System.out.println("Child class method");
    }
}
```

2. **Access Parent Class Variables**:
   - You can use `super` to access a field defined in the parent class if it is hidden by a field of the same name in the child class.

```
class Parent {
    int num = 100;
}

class Child extends Parent {
    int num = 200;

    void display() {
        System.out.println("Parent class variable: " + su
per.num); // Accesses parent class variable
        System.out.println("Child class variable: " + nu
m);
```

```
    }
}
```

3. **Invoke Parent Class Constructor**:
    - You can use `super()` to call the parent class constructor. This is useful when you want to initialize some properties in the parent class.

```
class Parent {
    Parent() {
        System.out.println("Parent class constructor");
    }
}

class Child extends Parent {
    Child() {
        super(); // Calls the parent class constructor
        System.out.println("Child class constructor");
    }
}
```

By using the `super` keyword, you can ensure that the child class has the ability to interact with and utilize the functionality provided by its parent class.

27) What are the 4 pillars of OOP? Explain each one.

The four pillars of Object-Oriented Programming (OOP) are:

1. **Encapsulation**:
    - Encapsulation is the process of wrapping data (variables) and code (methods) together as a single unit. It restricts direct access to some of an object's components, which can prevent the accidental modification of data. This is typically achieved through the use of access modifiers like private, protected, and public.
    - **Example**: In a class, private variables can only be accessed or modified through public getter and setter methods.

2. **Inheritance**:
    - Inheritance allows one class (the child or subclass) to inherit the fields and methods of another class (the parent or superclass). This promotes code reuse and establishes a natural hierarchy between classes.
    - **Example**: If you have a class `Animal` with a method `makeSound()`, a subclass `Dog` can inherit `Animal` and use the `makeSound()` method, while also adding its own methods or fields.
3. **Polymorphism**:
    - Polymorphism allows methods to do different things based on the object it is acting upon, even though they share the same name. It can be achieved through method overloading (same method name, different parameters) and method overriding (subclass provides a specific implementation of a method already defined in its superclass).
    - **Example**: A method `draw()` can be used for different shapes like circles, squares, and triangles, each having its own implementation of `draw()`.
4. **Abstraction**:
    - Abstraction involves hiding the complex implementation details and showing only the essential features of an object. It helps in reducing programming complexity and effort by providing a simplified model of the real-world entity.
    - **Example**: An abstract class `Vehicle` may have an abstract method `move()`. The subclasses `Car` and `Bike` will provide specific implementations of the `move()` method.

28)What are the access modifiers in Java? Explain them

Access Modifiers in Java:

1. **Public:**
    - **Scope:** Accessible from any other class.
    - **Usage:** Classes, methods, and fields can be declared as public.
    - **Example:**

```
public class MyClass {
    public int myField;
    public void myMethod() {}
}
```

2. **Protected:**
    - **Scope:** Accessible within the same package and by subclasses in different packages.
    - **Usage:** Methods and fields can be declared as protected.
    - **Example:**

```
class MyClass {
    protected int myField;
    protected void myMethod() {}
}
```

3. **Default (Package-Private):**
    - **Scope:** Accessible only within the same package.
    - **Usage:** No keyword is used; the absence of an access modifier implies default access.
    - **Example:**

```
class MyClass {
    int myField; // default access
    void myMethod() {} // default access
}
```

4. **Private:**
    - **Scope:** Accessible only within the same class.
    - **Usage:** Methods and fields can be declared as private.
    - **Example:**

```
class MyClass {
    private int myField;
    private void myMethod() {}
}
```

Summary:

- **Public:** Highest level of accessibility, available to all classes.
- **Protected:** Accessible within the same package and subclasses.
- **Default (Package-Private):** Accessible only within the same package.
- **Private:** Lowest level of accessibility, available only within the same class.

29) What is the difference between method overloading and overriding?

Method Overloading:

- **Definition**: Method overloading occurs when multiple methods in the same class have the same name but different parameters (different type, number, or both).
- **Compile-time Polymorphism**: Overloading is an example of compile-time polymorphism.
- **Return Type**: The return type can be the same or different.
- **Inheritance**: Overloading does not depend on inheritance; it can happen within the same class.
- **Example**:

```
public class Example {
    void display(int a) {
        System.out.println("Argument: " + a);
    }

    void display(int a, int b) {
        System.out.println("Arguments: " + a + ", " + b);
```

```
    }
}
```

Method Overriding:

- **Definition**: Method overriding occurs when a method in a child class has the same name, return type, and parameters as a method in its parent class.
- **Runtime Polymorphism**: Overriding is an example of runtime polymorphism.
- **Return Type**: The return type must be the same or a subtype (covariant return type) of the overridden method.
- **Inheritance**: Overriding is related to inheritance; the method in the child class overrides the method in the parent class.
- **Example**:

```
class Parent {
    void display() {
        System.out.println("Parent display method");
    }
}

class Child extends Parent {
    @Override
    void display() {
        System.out.println("Child display method");
    }
}
```

In summary, method overloading is about having multiple methods with the same name but different parameters within the same class, while method overriding is about redefining a method in a child class that is already defined in its parent class.

30)Difference between extends and implements?

Difference between extends and implements?

1. **Extends**:

- **Usage**: Used to inherit from a class or an abstract class.
- **Syntax**: `class SubClass extends SuperClass`
- **Purpose**: It allows a class to inherit fields and methods from another class, enabling code reuse and establishing an "is-a" relationship.
- **Example**:

```
class Animal {
    void eat() {
        System.out.println("Eating");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Barking");
    }
}
```

2. **Implements**:
    - **Usage**: Used to implement an interface.
    - **Syntax**: `class ClassName implements InterfaceName`
    - **Purpose**: It allows a class to provide implementations for the abstract methods defined in an interface, ensuring that the class adheres to a specific contract.
    - **Example**:

```
interface Animal {
    void eat();
}

class Dog implements Animal {
    public void eat() {
        System.out.println("Eating");
```

```
    }
}
```

Key Differences:

- **Inheritance Type**:
  - `extends`: Establishes a subclass-superclass relationship.
  - `implements`: Establishes a class-interface relationship.
- **Multiple Inheritance**:
  - `extends`: Java does not support multiple inheritance for classes (a class can only extend one class).
  - `implements`: A class can implement multiple interfaces.
- **Method Implementation**:
  - `extends`: The subclass inherits or overrides methods from the superclass.
  - `implements`: The class must provide concrete implementations for all abstract methods declared in the interface.

31) What is the difference between an abstract class and an interface?

-An abstract class is a class that cannot be instantiated on its own and is intended to be inherited by other classes.

-It can have both abstract methods (methods declared without an implementation) and concrete methods (methods with an implementation)

-A class can only inherit from one abstract class.

• An interface is a contract that specifies a set of methods that must be implemented by any class that implements it.

• An interface cannot have any implementation code; all methods are abstract.

• A class can implement multiple interfaces.

32) What are the implicit modifiers for interface variables / methods?

Interface Variables:

- Interface variables are implicitly `public`, `static`, and `final`. This means that:
  - They can be accessed from anywhere (because they're `public`).
  - They belong to the interface itself, not to any instance of a class that implements the interface (because they're `static`).
  - They cannot be reassigned or changed once they're initialized (because they're `final`).

Interface Methods:

- Interface methods are implicitly `public` and `abstract`. This means that:
  - They can be accessed from anywhere (because they're `public`).
  - They must be implemented by any class that implements the interface (because they're `abstract`).

33) Can you overload / override a main method? static method? a private method? a default method? a protected method?

Overloading/Overriding a `main` method:

- You can overload the `main` method, but it won't be used as the entry point for your program. The JVM looks for a `main` method with a specific signature (`public static void main(String[] args)`) to start the program.
- You cannot override the `main` method in the classical sense, because it's a static method. However, you can define a `main` method in a subclass, but it will be a separate method from the one in the superclass.

Overloading/Overriding a `static` method:

- You can overload a `static` method, just like any other method.
- You cannot override a `static` method in the classical sense. If you define a `static` method with the same name and signature in a subclass, it will be a separate method from the one in the superclass. This is known as "hiding" the superclass method.

Overloading/Overriding a `private` method:

- You cannot overload or override a `private` method, because it's not visible outside the class where it's defined.
- If you define a method with the same name and signature as a `private` method in a superclass, it will be a separate method and won't be related to the superclass method.

Overloading/Overriding a `default` method (in an interface):

- You can override a `default` method in an interface by providing a different implementation in a class that implements the interface.
- You cannot overload a `default` method in an interface, because interfaces do not support method overloading.

Overloading/Overriding a `protected` method:

- You can overload a `protected` method, just like any other method.
- You can override a `protected` method in a subclass, as long as the subclass is in the same package as the superclass or is a subclass of the superclass.

34. What is the base class of all exceptions?

The base class of all exceptions in Java is `java.lang.Throwable`. However, the base class of all exceptions that can be caught and handled by the application is `java.lang.Exception`. `Throwable` is the superclass of `Exception` and `Error`.

35. Difference between checked and unchecked exceptions?

Checked exceptions are those that are checked at compile-time, meaning the compiler will check if the code handles or declares them. Unchecked exceptions are

not checked at compile-time and are typically used for programming errors.

Here are the key differences:

- **Checked exceptions**:
  - Are checked at compile-time.
  - Must be either caught or declared in the method signature using the `throws` keyword.
  - Typically represent errors that can be anticipated and recovered from, such as file not found or network connection errors.
  - Examples: `IOException`, `SQLException`, `FileNotFoundException`.
- **Unchecked exceptions**:
  - Are not checked at compile-time.
  - Do not need to be caught or declared in the method signature.
  - Typically represent programming errors, such as null pointer exceptions or array index out of bounds.
  - Examples: `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ClassCastException`.

36) Different ways of handling checked exceptions?

1. **Catch the exception**: Use a `try-catch` block to catch the exception and handle it.
2. **Declare the exception**: Use the `throws` keyword to declare the exception in the method signature

What is try-with-resources? What interface must the resource implement to use this feature?

Try-with-resources is a feature introduced in Java 7 that allows you to automatically close resources, such as files or connections, when they are no longer needed.

To use try-with-resources, the resource must implement the `AutoCloseable` interface, which has a single method `close()`. The `Closeable` interface, which

extends `AutoCloseable`, is also commonly used.

What is the difference between final, .finalize(), and finally?

- **final**: A keyword used to declare a variable, method, or class that cannot be modified or overridden.
- **finalize()**: A method that is called by the garbage collector before an object is garbage collected. It is used to release system resources, such as file handles or sockets.
- **finally**: A block of code that is executed after a `try-catch` block, regardless of whether an exception was thrown or not. It is used to release resources, such as closing files or connections.

throw vs throws vs Throwable?

- **throw**: A keyword used to throw an exception explicitly.
- **throws**: A keyword used to declare an exception in a method signature.
- **Throwable**: The base class of all exceptions and errors in Java

Multi-catch block - can you catch more than one exception in a single catch block?

Yes, you can catch more than one exception in a single catch block using the `|` operator. This is called a multi-catch block.

```
try {
// code that may throw an exception
} catch (IOException | SQLException e) {
// handle the exception
}
```

BufferReader -Why this if we have just inputstream or filereader what is need of buffer reader

`BufferedReader` is a wrapper around `Reader` objects, such as `FileReader` or `InputStreamReader`, that provides several benefits:

1. **Buffering**: The primary purpose of `BufferedReader` is to improve performance by buffering the input. When you read data from a file or input stream, the operating system typically reads data in blocks, not one character at a time. `BufferedReader` takes advantage of this by reading a larger block of data (the buffer) and storing it in memory. This allows for faster access to the data because subsequent reads can be satisfied from the buffer instead of requiring a disk access.
2. **Efficient Reading**: `BufferedReader` provides methods like `readLine()` that allow you to read a line of text at a time. This is more efficient than reading individual characters and checking for newline characters yourself.
3. **Convenience Methods**: `BufferedReader` offers additional convenience methods, such as `skip()`, `mark()`, `reset()`, and `ready()`, which can be useful in certain situations.

what is serialization?

Ability of object to be converted into stream of bytes transfer over network or store in file and then recreating or restoring its form in object is called serialization.

Logback Architecture

Logback is a popular logging framework for Java-based applications. It's designed to be a successor to the popular log4j framework. Logback's architecture is based on a modular design, which makes it flexible, customizable, and highly configurable.

Here's an overview of the Logback architecture:

1. **Logger**: The Logger is the core component of Logback. It's responsible for capturing log events and forwarding them to Appenders. Loggers are usually named and can be configured to log at different levels.
2. **Appenders**: Appenders are responsible for outputting log events to various destinations, such as files, consoles, or networks. Logback provides several built-in Appenders, including FileAppender, ConsoleAppender, and SocketAppender.

3. **Layout**: The Layout component is responsible for formatting log events into a human-readable format. Logback provides several built-in Layouts, including PatternLayout and XMLLayout.
4. **Filters**: Filters are used to selectively discard or accept log events based on certain criteria, such as log level, marker, or message content.
5. **Context**: The Context component provides a way to configure Logback and manage its lifecycle.

Logging Levels

Logback provides the following logging levels, in order of increasing severity:

1. **TRACE**: The TRACE level is used for debugging purposes and is typically used to log fine-grained information about the application's behavior.
2. **DEBUG**: The DEBUG level is used for debugging purposes and is typically used to log information about the application's behavior that's useful for troubleshooting.
3. **INFO**: The INFO level is used to log information about the application's behavior that's useful for monitoring and auditing.
4. **WARN**: The WARN level is used to log potential problems or unexpected events that might not be errors but are worth noting.
5. **ERROR**: The ERROR level is used to log serious errors or exceptions that prevent the application from functioning correctly.

Difference between scrum and kanban

Scrum and Kanban: A Comparison

Scrum and Kanban are two popular Agile frameworks used in software development, IT, and other industries. While both frameworks share some similarities, they have distinct differences in their approach, principles, and practices.

Scrum

Scrum is a structured framework that emphasizes teamwork, accountability, and iterative progress toward well-defined goals. It's based on three pillars:

1. **Transparency**: Clear goals, progress, and challenges are visible to all stakeholders.
2. **Inspection**: Regular inspections of progress and adaptation to changes.
3. **Adaptation**: Flexibility to respond to changes and improve processes.

Scrum framework consists of:

- **Sprint**: A fixed-duration iteration (usually 2-4 weeks) to complete a set of work items.
- **Scrum Team**: Cross-functional team of 3-9 members, including a **Product Owner**, **Scrum Master**, and **Development Team**.
- **Product Backlog**: Prioritized list of features or user stories to be developed.
- **Sprint Planning**: Team plans and commits to a set of work items for the upcoming sprint.
- **Daily Scrum**: Daily meeting to review progress, discuss obstacles, and plan work for the day.
- **Sprint Review**: Review of completed work at the end of the sprint.
- **Sprint Retrospective**: Team reflection on the sprint process to identify improvements.

Kanban

Kanban is a visual system for managing work, emphasizing continuous flow and limiting work in progress. It's based on the following principles:

1. **Visualize the workflow**: Map the workflow to understand the process and identify bottlenecks.
2. **Limit work in progress (WIP)**: Set limits on the amount of work in progress to maintain a smooth flow.
3. **Focus on flow**: Prioritize the smooth flow of work items through the system.
4. **Continuous improvement**: Regularly review and improve the process.

Kanban framework consists of:

- **Kanban Board**: Visual representation of the workflow, showing columns for different stages (e.g., To-Do, In Progress, Done).
- **Work Items**: Tasks, features, or user stories to be developed.
- **WIP Limits**: Limits on the number of work items in each stage to maintain a smooth flow.
- **Pull-based workflow**: Team members pull work items into their workflow as capacity allows.
- **Continuous Delivery**: Regular delivery of working software to stakeholders.

Key Differences

Here are the main differences between Scrum and Kanban:

- **Structure**: Scrum has a more structured approach with sprints, roles, and ceremonies, while Kanban is more flexible and adaptive.
- **Iterations**: Scrum uses fixed-duration sprints, while Kanban uses a continuous flow of work items.
- **Roles**: Scrum has defined roles (Product Owner, Scrum Master, Development Team), while Kanban does not have specific roles.
- **Prioritization**: Scrum prioritizes work items through the Product Backlog, while Kanban prioritizes work items through the Kanban Board and WIP limits.
- **Meetings**: Scrum has regular meetings (Sprint Planning, Daily Scrum, Sprint Review, Sprint Retrospective), while Kanban does not have prescribed meetings.

When to Choose Scrum or Kanban

- **Scrum**: Suitable for teams that need a structured approach, have a clear understanding of the work to be done, and can commit to sprints.
- **Kanban**: Suitable for teams that need a more flexible approach, have a high degree of uncertainty, or want to focus on continuous flow and delivery.

Ultimately, the choice between Scrum and Kanban depends on your team's specific needs, preferences, and goals. Both frameworks can be effective in improving collaboration, productivity, and delivery quality.

What is the Waterfall Model?

The Waterfall Model is a traditional software development process that follows a linear and sequential approach. It is a structured methodology that divides the software development process into distinct phases, with each phase completed before moving on to the next one.

The phases of the Waterfall Model are:

1. **Requirements Gathering**: Collecting and documenting the requirements of the project.
2. **Analysis**: Breaking down the requirements into smaller, manageable parts.
3. **Design**: Creating a detailed design of the system, including architecture, components, and interfaces.
4. **Implementation** (Coding): Writing the code for the system.
5. **Testing**: Verifying that the system meets the requirements and works as expected.
6. **Deployment**: Releasing the system to the end-users.
7. **Maintenance**: Making updates, fixes, and enhancements to the system.

Advantages of the Waterfall Model:

1. **Easy to manage**: The Waterfall Model is easy to understand and manage, as each phase has a clear start and end point.
2. **Predictable**: The model provides a clear timeline and budget for the project, making it easier to plan and estimate resources.
3. **Sequential**: Each phase is completed before moving on to the next one, reducing the risk of errors and misunderstandings.
4. **Documentation**: The model emphasizes documentation, which helps in maintaining a record of the project's progress and decisions.

Disadvantages of the Waterfall Model:

1. **Inflexible**: The model does not allow for changes or flexibility once a phase is completed.
2. **High risk**: If a problem is discovered in a later phase, it can be costly and time-consuming to go back and fix it.

3. **No working software**: The model does not provide a working software until the end of the project, which can be a long time.
4. **Customer involvement**: The model does not involve the customer throughout the development process, which can lead to misunderstandings and miscommunication.

What is SDLC?

SDLC stands for Software Development Life Cycle. It is a framework that outlines the stages involved in planning, creating, testing, and delivering software applications. SDLC provides a structured approach to software development, ensuring that the software is developed on time, within budget, and meets the required quality standards.

The SDLC process typically includes the following stages:

1. **Planning**: Defining project scope, goals, and timelines.
2. **Analysis**: Gathering and documenting requirements.
3. **Design**: Creating a detailed design of the system.
4. **Implementation** (Coding): Writing the code for the system.
5. **Testing**: Verifying that the system meets the requirements and works as expected.
6. **Deployment**: Releasing the system to the end-users.
7. **Maintenance**: Making updates, fixes, and enhancements to the system.

What are Agile Manifesto Principles?

The Agile Manifesto is a document that outlines the values and principles of the Agile software development methodology. The Agile Manifesto was created in 2001 by a group of software developers as a response to the traditional, rigid approaches to software development.

The Agile Manifesto values are:

1. **Individuals and interactions**: People and communication are more important than processes and tools.

2. **Working software**: Working software is more important than comprehensive documentation.
3. **Customer collaboration**: Collaboration with customers is more important than contract negotiation.
4. **Responding to change**: Responding to change is more important than following a plan.

What is Maven?

Maven is **an open-source build automation and project management tool that helps developers build, publish, and deploy projects**

What is Scrum?

Scrum is an **Agile** framework for managing and completing complex projects. It is a popular approach used in software development, but it can be applied to any project that requires iterative development and collaboration between cross-functional teams.

The Scrum framework consists of three roles:

1. **Product Owner**: responsible for defining and prioritizing the product backlog, which is a list of features, requirements, and bugs that need to be addressed.
2. **Development Team**: responsible for delivering a potentially shippable product increment at the end of each sprint.
3. **Scrum Master**: responsible for facilitating the Scrum process, removing impediments, and ensuring that the team follows the Scrum framework.

What is Type Casting?

Type casting is a process in programming where a value of one data type is converted into another data type. This is often necessary when a function or

operation requires a specific data type, but the value you want to use is of a different type.

There are two types of type casting:

1. **Implicit Type Casting**: This is an automatic type conversion done by the compiler. For example, in many programming languages, a smaller data type can be implicitly converted to a larger data type. For example, an `int` can be implicitly converted to a `float`.
2. **Explicit Type Casting**: This is a manual type conversion done by the programmer using a cast operator. For example, in C-style languages, you can use the `(type)` operator to explicitly cast a value to a specific type.

What is Stream API?

Stream API is a way **to express and process collections of objects**. Enable us to perform operations like filtering, mapping, reducing, and sorting

The Java Stream API is a powerful feature introduced in Java 8 that allows you to process data in a declarative way. It provides a more concise and expressive way to perform complex data processing operations, making your code more readable and maintainable

Stream API Components

1. **Source**: The origin of the data (e.g., collections, arrays, files).
2. **Intermediate operations**: Methods that transform or filter the data (e.g., `map()`, `filter()`).
3. **Terminal operations**: Methods that produce a result or side effect (e.g., `forEach()`, `collect()`).

What is a Lambda Function?

A lambda function is an anonymous function that can be defined inline within a larger expression. It consists of three parts:

1. **Input parameters**: The variables that are passed to the lambda function.
2. **Lambda operator**: The `>` symbol that separates the input parameters from the lambda body.
3. **Lambda body**: The code that is executed when the lambda function is invoked.

What are Sprint Durations?

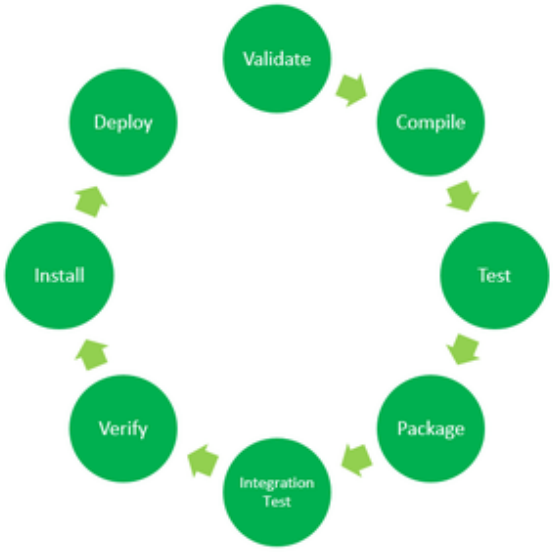
1. **1-2 weeks**: Short sprints are ideal for teams that need to respond quickly to changing requirements or have a high-priority backlog.
2. **2-4 weeks**: Medium-length sprints are suitable for teams that need to balance speed with stability and have a moderate-priority backlog.
3. **4-6 weeks**: Long sprints are best for teams that need to tackle complex tasks or have a low-priority backlog.

What are Maven Commands?

Basic Commands

1. `mvn clean`: Deletes the target directory and any generated files.
2. `mvn compile`: Compiles the source code.
3. `mvn package`: Packages the compiled code into a JAR or WAR file.
4. `mvn install`: Installs the packaged JAR or WAR file into the local repository.
5. `mvn deploy`: Deploys the packaged JAR or WAR file to a remote repository.

Maven Lifecycle:



- **Validate:** This step validates if the project structure is correct. For example – It checks if all the dependencies have been downloaded and are available in the local repository.
- **Compile:** It compiles the source code, converts the .java files to .class, and stores the classes in the target/classes folder.
- **Test:** It runs unit tests for the project.
- **Package:** This step packages the compiled code in a distributable format like JAR or WAR.
- **Integration test:** It runs the integration tests for the project.
- **Verify:** This step runs checks to verify that the project is valid and meets the quality standards.
- **Install:** This step installs the packaged code to the local Maven repository.
- **Deploy:** It copies the packaged code to the remote repository for sharing it with other developers.

Explain Git Pull?

Git Pull is a command used to fetch and merge changes from a remote repository into the local repository. It's a convenient way to update your local repository with the latest changes from the remote repository.

How Git Pull Works

When you run `git pull`, Git performs the following steps:

1. **Fetch**: Git fetches the latest changes from the remote repository. This involves downloading the latest commits, branches, and tags from the remote repository.
2. **Merge**: Git merges the fetched changes into the local repository. This involves integrating the changes from the remote repository into the local repository.

Explain Git Origin?

In Git, `origin` is a special name given to the remote repository from which a local repository was cloned. It's a convention that Git uses to identify the original repository from which a local copy was made.

What is Git Origin?

When you clone a repository using `git clone`, Git automatically sets up a remote repository named `origin` that points to the original repository. This allows you to easily interact with the original repository, such as fetching updates, pushing changes, and tracking branches.

== and equals diff

`==` Operator

The `==` operator is used to compare the memory locations of two objects. It checks whether both objects point to the same location in memory. In other words, it checks whether the two objects are the same instance.

`equals()` Method

The `equals()` method is used to compare the contents of two objects. It checks whether the two objects have the same value, regardless of whether they are the same instance or not.

String s1 = new String("Hello");
String s2 = new String("Hello");

System.out.println(s1 == s2); // false
System.out.println(s1.equals(s2)); // true

What is the Staging Area?

The Staging Area is a virtual area where you stage changes to your files before committing them to the Git repository. It's a buffer zone where you can review and prepare your changes before making them permanent.

Think of the Staging Area as a "draft" area where you can:

- Add new files or changes to be committed
- Remove files or changes that should not be committed
- Modify files or changes before committing them

Merge Conflict? How to manage it?

A merge conflict occurs when two or more developers make changes to the same code file, and Git is unable to automatically merge those changes. This happens when the changes are conflicting, meaning they cannot be merged without manual intervention.

What causes Merge Conflicts?

Merge conflicts can occur due to various reasons, including:

- **Simultaneous changes**: Multiple developers make changes to the same file simultaneously.
- **Conflicting changes**: Changes made by different developers are incompatible with each other.
- **File renames or moves**: Files are renamed or moved, causing conflicts when merging.

Sdlc ceremonies

1. Sprint Planning

- Purpose: Define the work to be done during the upcoming sprint.
- Attendees: Development team, product owner, and stakeholders.
- Outcome: A clear understanding of the sprint goals, tasks, and priorities.

2. Daily Stand-up (Daily Scrum)

- Purpose: Review progress, discuss obstacles, and plan the day's work.
- Attendees: Development team.
- Outcome: Team members are aware of each other's progress, and any obstacles are addressed.

3. Sprint Review (Demo)

- Purpose: Showcase the work completed during the sprint.
- Attendees: Development team, product owner, and stakeholders.
- Outcome: Feedback is gathered, and the team receives recognition for their work.

4. Sprint Retrospective

- Purpose: Reflect on the sprint, identify improvements, and implement changes.
- Attendees: Development team.

- Outcome: The team identifies areas for improvement and implements changes to increase efficiency and effectiveness.

1. Functional Interface

A functional interface is an interface that has only one abstract method (SAM - Single Abstract Method). It is used to represent a function or a lambda expression. Functional interfaces are used extensively in Java 8's lambda expressions and method references.

2. How to switch branch in Git?

To switch to a different branch in Git, you can use the `git checkout` command followed by the name of the branch you want to switch to.

4. What are exceptions? How to handle? Types? Examples?

Exceptions are runtime errors that occur during the execution of a program. They can be handled using try-catch blocks.

Types of exceptions:

- **Checked exceptions**: These are exceptions that are checked at compile-time. Examples: IOException, SQLException.
- **Unchecked exceptions**: These are exceptions that are not checked at compile-time. Examples: NullPointerException, ArithmeticException.

5. Autoboxing and Unboxing

Autoboxing is the automatic conversion of a primitive type to its corresponding wrapper class. Unboxing is the automatic conversion of a wrapper class to its corresponding primitive type.

6. Daily Scrum use?

Daily Scrum is a meeting in Agile development where team members discuss:

- What they worked on the previous day
- What they plan to work on that day
- Any obstacles or challenges they are facing

The purpose of Daily Scrum is to:

- Improve communication among team members
- Identify and address obstacles early
- Increase team velocity and productivity

7. Git commands from basic

Here are some basic Git commands:

- `git init`: Initializes a new Git repository
- `git add <file>`: Stages a file for the next commit
- `git commit -m "<message>"`: Commits changes with a message
- `git log`: Displays a log of all commits
- `git branch <branch-name>`: Creates a new branch
- `git checkout <branch-name>`: Switches to a different branch
- `git merge <branch-name>`: Merges changes from another branch
- `git remote add <name> <url>`: Adds a remote repository
- `git push`: Pushes changes to a remote repository

- `git pull`: Pulls changes from a remote repository

8. POM (Project Object Model)

A POM (Project Object Model) is an XML file used in Maven to define a project's structure, dependencies, and build process. It contains information such as:

- Project name and version
- Dependencies and their versions
- Build plugins and their configurations
- Project repositories and distribution management