Cheat sheet - Summarization of OS in 4th sem

Operating Systems (Kalinga Institute of Industrial Technology)

Java cheat sheet for wt

1. Java Final Keyword:

The final keyword in Java is used to restrict the user from changing the value of a variable, declaring a method that cannot be overridden by child classes or declaring a class that cannot be inherited. Once the final keyword is used on any entity, it cannot be changed.

Example:

```
public class Example {
  public final int NUMBER = 10; // final variable
  public final void display() { // final method
    System.out.println("This is a final method.");
  }
}
public class SubExample extends Example {
  public void display() { // compile-time error
    System.out.println("This method cannot be overridden.");
  }
}
```

2. Java Static Keyword:

The static keyword in Java is used to define a class-level variable or method that can be accessed without creating an object of the class. The static variable or method belongs to the class rather than a specific instance of the class.

Example:

```
public class Example {
  public static int NUMBER = 10; // static variable
  public static void display() { // static method
    System.out.println("This is a static method.");
  }
}
public class SubExample extends Example {
  public void show() {
    System.out.println(NUMBER); // accessing static variable
    display(); // accessing static method
  }
}
```

3. Java Super Keyword:
The super keyword in Java is used to access the parent class constructor, parent class methods, and parent class variables. It is mostly used in inheritance where a child class is derived from a parent class.

Example:
```
public class ParentClass {
  public void display() {
    System.out.println("This is a parent class method.");
  }
}
public class ChildClass extends ParentClass {
  public void display() {
    super.display(); // accessing parent class method
    System.out.println("This is a child class method.");
  }
}
```

4. Constructors in java:-
In Java, a constructor is a special method that is used to initialize objects of a class. It is called when an object of a class is created, and its main purpose is to set the initial values of the object's instance variables. Constructors have the same name as the class in which they are defined, and they don't have a return type.

There are three types of constructors in Java:

Default constructor: A default constructor is a constructor that takes no arguments. If a class doesn't have any constructor defined explicitly, Java automatically provides a default constructor for that class. It initializes all instance variables to their default values (e.g., 0 for integers, false for booleans, null for objects).
Example:
```
public class Person {
    String name;
    int age;
    // Default constructor
    public Person() {
    }
}
```

Parameterized constructor: A parameterized constructor is a constructor that takes one or more arguments. It is used to set the initial values of the object's instance variables based on the values passed as arguments.
Example:

```
public class Person {
   String name;
   int age;
   // Parameterized constructor
   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
}
```

Copy constructor: A copy constructor is a constructor that takes an object of the same class as a parameter and creates a new object with the same values as the parameter object. It is used to create a copy of an existing object.
Example:

```
public class Person {
   String name;
   int age;
   // Copy constructor
   public Person(Person other) {
      this.name = other.name;
      this.age = other.age;
   }
}
```

Constructors are important in Java because they ensure that objects are properly initialized before they are used. They also make it easy to create new objects with different initial values, and to create copies of existing objects.

5. Java achieves runtime polymorphism through method overriding. Method overriding allows a subclass to provide its own implementation of a method that is already provided by its parent class. In order for method overriding to occur, the method name, parameter types, and

return type must be exactly the same in the subclass as in the parent class. The subclass must also have an IS-A relationship with the parent class.

Here is an example that demonstrates runtime polymorphism through method overriding in Java:

```
class Animal {
  public void makeSound() {
    System.out.println("Some sound");
  }
}
class Dog extends Animal {
  @Override
  public void makeSound() {
    System.out.println("Woof");
  }
}

class Cat extends Animal {
  @Override
  public void makeSound() {
    System.out.println("Meow");
  }
}

public class Main {
  public static void main(String[] args) {
    Animal animal1 = new Dog();
    Animal animal2 = new Cat();
    animal1.makeSound();
    animal2.makeSound();
  }
}
```

In this example, we have an Animal class with a makeSound() method. We also have two subclasses Dog and Cat that override the makeSound() method with their own implementations.

In the main() method, we create two Animal objects, one of type Dog and the other of type Cat. When we call the makeSound() method on

each object, Java will use the appropriate implementation of the method based on the actual type of the object at runtime. This is known as dynamic method dispatch.

So, when we run this program, it will output:
Woof
Meow

This is an example of runtime polymorphism because the makeSound() method is called on the Animal objects at runtime, and Java is able to determine which implementation of the method to use based on the actual type of the object.

6. Arrays in Java and C share some similarities but also have some differences.

One major difference is that in Java, arrays are objects, whereas in C, arrays are just blocks of memory allocated for a certain type of data. Java arrays are implemented as objects and provide many built-in methods, such as length(), which returns the number of elements in the array.

Here's an example to illustrate the difference between Java and C arrays:

Java:
```
int[] arr = new int[5]; // create an integer array of size 5
arr[0] = 1; // set the first element to 1
arr[1] = 2; // set the second element to 2
System.out.println(arr.length); // prints 5
```

C:
```
int arr[5]; // create an integer array of size 5
arr[0] = 1; // set the first element to 1
arr[1] = 2; // set the second element to 2
int len = sizeof(arr)/sizeof(arr[0]); // calculate the length of the array
printf("%d", len); // prints 5
```

In Java, we declare an array using the new keyword and specify its size. We can then access the elements of the array using square brackets [].

We can also use the built-in method length() to get the length of the array.

In C, we declare an array by specifying its size in square brackets []. We can access the elements of the array using square brackets []. To get the length of the array, we can calculate the size of the array in bytes using sizeof() and divide it by the size of one element of the array.

One key difference is that in Java, arrays are objects, while in C, arrays are a type of data structure. This means that arrays in Java can have methods and be manipulated using object-oriented principles, while in C, arrays are typically manipulated using pointers.

One key difference between these two examples is the placement of the square brackets. In Java, the brackets come after the data type, while in C, they come after the variable name.

Another difference between Java and C arrays is that in Java, arrays have a fixed size that is specified when the array is created. In C, arrays can have a fixed size, but they can also be dynamically allocated at run-time using functions like malloc() and calloc().

Overall, while there are some similarities between arrays in Java and C, there are also some significant differences that reflect the different programming paradigms of the two languages.

7. Characteristics of Java:-

Java is a popular programming language used for developing a wide range of applications. Some of the key characteristics of Java include:

Object-Oriented: Java is an object-oriented programming language, which means it focuses on creating objects and their interactions to build software applications.

Platform Independent: One of the most significant features of Java is that it is platform-independent, which means that a Java program can run on any platform, regardless of the hardware or software environment.

Simple: Java is designed to be simple and easy to learn, with a syntax that is similar to that of C++ but with fewer low-level features.

Robust: Java is a robust language with built-in mechanisms for handling errors, exceptions, and memory management. It also includes automatic garbage collection, which frees developers from having to manage memory manually.

Secure: Java is designed with security in mind and includes features like sandboxing, which allows applications to run in a restricted environment and prevents them from accessing system resources.

Multithreaded: Java supports multithreading, which means that multiple threads of execution can run simultaneously within the same program, improving performance and responsiveness.

Portable: Because Java is platform-independent, it is highly portable, which means that applications written in Java can be easily moved from one platform to another without having to recompile the code.

Distributed: Java includes support for distributed computing, which means that applications can be built that run on multiple systems and communicate with each other over a network.

Overall, these characteristics make Java a popular language for building robust, secure, and scalable applications that can run on a wide range of platforms and devices.

8. Dynamic method dispatch is a mechanism in Java that enables a program to call an overridden method of an object at runtime, rather than at compile time. This mechanism is also known as runtime polymorphism or late binding.

When a subclass overrides a method of its superclass, and an object of the subclass is assigned to a superclass reference variable, then the overridden method of the subclass is called through the superclass reference variable. This is possible because the method call is resolved at runtime, based on the actual type of the object being referred to, rather than the declared type of the reference variable.

For example, consider the following code:

```
class Animal {
   public void makeSound() {
      System.out.println("Animal is making a sound");
   }
}
class Cat extends Animal {
   public void makeSound() {
      System.out.println("Cat is meowing");
   }
}
class Dog extends Animal {
   public void makeSound() {
      System.out.println("Dog is barking");
   }
}
public class Test {
   public static void main(String[] args) {
      Animal animal1 = new Cat();
      animal1.makeSound();  // Output: Cat is meowing

      Animal animal2 = new Dog();
      animal2.makeSound();  // Output: Dog is barking
   }
}
```

In this example, the Animal class has a method called makeSound(). The Cat and Dog classes override this method to provide their own implementation. In the main() method, we create an object of the Cat class and assign it to an Animal reference variable. Similarly, we create an object of the Dog class and assign it to another Animal reference variable. When we call the makeSound() method on these reference variables, the overridden methods of the Cat and Dog classes are called, respectively. This is an example of dynamic method dispatch in Java.

9. Abstract Class:-
In Java, an abstract class is a class that is declared abstract and cannot be instantiated. It can only be used as a base class for other classes that extend it. Abstract classes provide a way to define a common interface for a set of subclasses. It is like a blueprint for the subclasses, providing a skeleton or template for them to follow.

An abstract class may contain both abstract and concrete methods. Abstract methods are declared but not defined, while concrete methods are defined with an implementation. Subclasses of an abstract class must implement all abstract methods of the parent class.

Some of the characteristics of abstract classes are:

An abstract class cannot be instantiated directly; it can only be used as a superclass for other classes.
An abstract class may contain abstract and/or concrete methods.
Abstract methods must be implemented by subclasses.
An abstract class may contain instance variables.
Abstract classes can have constructors, but they cannot be used to instantiate objects directly.
A class can extend only one abstract class at a time, but it can implement multiple interfaces.
Example:

```
public abstract class Animal {
  private String name;
  public Animal(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public abstract void makeSound();
}
public class Dog extends Animal {
  public Dog(String name) {
    super(name);
  }
  public void makeSound() {
    System.out.println("Woof!");
```

```
  }
}
public class Cat extends Animal {
  public Cat(String name) {
    super(name);
  }
  public void makeSound() {
    System.out.println("Meow!");
  }
}
```

In the above example, Animal is an abstract class with an abstract method makeSound(). The Dog and Cat classes extend the Animal class and implement the makeSound() method.

In Java, a class is a blueprint or template for creating objects. It defines the properties and behavior of objects that belong to that class.

For example, let's say we want to create a class called Car which represents a car in a program. The Car class would have properties such as make, model, year, color, and methods such as start(), stop(), accelerate(), brake(), etc.

Once we define the Car class, we can create objects (or instances) of the Car class. Each object will have its own set of properties and can perform its own set of behaviors. For example, we can create an object of the Car class called myCar, with properties such as make = "Toyota", model = "Camry", year = 2022, color = "Red". We can then call the methods of the myCar object, such as myCar.start(), myCar.accelerate(), etc.

In summary, a class defines the properties and behaviors of a group of objects, while an object is an instance of a class with its own unique set of properties and behaviors.

10. OOPs in java:-

OOPs stands for Object-Oriented Programming, which is a programming paradigm based on the concept of objects. In Java, everything is considered to be an object, and the program is designed to manipulate these objects.

The four pillars of OOPs in Java are:

Encapsulation: It is a mechanism of wrapping data and code together as a single unit. It is used to protect the data from unauthorized access by providing access only through the methods of the class. In Java, we use access modifiers like public, private, and protected to provide encapsulation.
Example:

```
public class BankAccount {
  private double balance;
  public void deposit(double amount) {
    balance += amount;
  }
  public void withdraw(double amount) {
    if (balance >= amount) {
      balance -= amount;
    }
  }
  public double getBalance() {
    return balance;
  }
}
```

Inheritance: It is a mechanism of creating a new class from an existing class. The new class inherits the properties and methods of the existing class. It helps in reusing the code and also promotes code extensibility.
Example:

```
public class Animal {
  public void eat() {
    System.out.println("Animal is eating");
  }
}
public class Dog extends Animal {
```

```
  public void bark() {
    System.out.println("Dog is barking");
  }
}
public class Main {
  public static void main(String[] args) {
    Dog dog = new Dog();
    dog.eat();  // Output: Animal is eating
    dog.bark(); // Output: Dog is barking
  }
}
```

Polymorphism: It is a mechanism of performing a single action in different ways. In Java, polymorphism is achieved through method overloading and method overriding.

Example:

```
public class Animal {
  public void makeSound() {
    System.out.println("Animal is making a sound");
  }
}
public class Dog extends Animal {
  @Override
  public void makeSound() {
    System.out.println("Dog is barking");
  }
}
public class Cat extends Animal {
  @Override
  public void makeSound() {
    System.out.println("Cat is meowing");
  }
}
public class Main {
  public static void main(String[] args) {
    Animal animal1 = new Dog();
    Animal animal2 = new Cat();
    animal1.makeSound(); // Output: Dog is barking
    animal2.makeSound(); // Output: Cat is meowing
  }
```

}

Abstraction: It is a mechanism of hiding the implementation details and showing only the essential features of the object. In Java, we use abstract classes and interfaces to achieve abstraction.
Example:

```
public abstract class Shape {
  public abstract double area();
}
public class Circle extends Shape {
  private double radius;
  public Circle(double radius) {
    this.radius = radius;
  }
  @Override
  public double area() {
    return Math.PI * radius * radius;
  }
}
public class Rectangle extends Shape {
  private double length;
  private double width;
  public Rectangle(double length, double width) {
    this.length = length;
    this.width = width;
  }
  @Override
  public double area() {
    return length * width;
  }
}
public class Main {
  public static void main(String[] args) {
    Shape shape1 = new Circle(5);
    Shape shape2 = new Rectangle(5, 10);
    System.out.println("Area of Circle: " + shape1.area());    // Output:
Area of Circle: 78.53981633974483
    System.out.println("Area of Rectangle: " + shape2.area()); // Output
```

Encapsulation is one of the fundamental concepts of object-oriented programming (OOP) that describes the principle of data hiding. It refers to the practice of hiding internal data and methods of an object from the outside world and providing access to them only through publicly exposed methods, which are also known as getters and setters. Encapsulation allows the programmer to implement the concept of data abstraction by restricting access to the implementation details of an object and exposing only what is necessary.

In Java, encapsulation is achieved through the use of access modifiers, such as private, public, and protected. Private members of a class are not accessible outside the class, while public members can be accessed from anywhere. Protected members can be accessed by classes in the same package or by derived classes.

The following is an example of encapsulation in Java:

```
public class BankAccount {
   private double balance;
   public void deposit(double amount) {
      balance += amount;
   }
   public void withdraw(double amount) {
      balance -= amount;
   }
   public double getBalance() {
      return balance;
   }
}
```

In this example, the balance variable is marked as private, which means it is not accessible outside the BankAccount class. The deposit and withdraw methods are public, which means they can be accessed from outside the class. However, they modify the balance variable indirectly through method calls, rather than directly accessing it. The getBalance method is also public, but it only returns the value of the balance variable and does not modify it. This is an example of encapsulation, where the implementation details of the BankAccount class are hidden from the outside world, and access to the balance variable is controlled through publicly exposed methods.

11. INHERITANCE in java:-
Inheritance is a mechanism in object-oriented programming where a class can inherit properties and behaviors from a parent class. The child class or subclass can reuse the code from the parent class or superclass, and also add new methods and properties specific to the child class.

There are several types of inheritance in Java:

Single inheritance: A subclass extends a single superclass. For example:

```
class Animal {
   void eat() {
      System.out.println("Animal is eating");
   }
}
class Dog extends Animal {
   void bark() {
      System.out.println("Dog is barking");
   }
}
```

In this example, Dog is a subclass of Animal. It inherits the eat() method from Animal and adds a new method bark() specific to Dog.

Multi-level inheritance: A subclass extends another subclass. For example:

```
class Animal {
   void eat() {
      System.out.println("Animal is eating");
   }
}
class Mammal extends Animal {
   void move() {
      System.out.println("Mammal is moving");
   }
}
class Dog extends Mammal {
   void bark() {
      System.out.println("Dog is barking");
   }
}
```

In this example, Mammal is a subclass of Animal, and Dog is a subclass of Mammal. Dog inherits both eat() and move() methods from its ancestors.

Hierarchical inheritance: Multiple subclasses extend a single superclass. For example:

```
class Animal {
   void eat() {
      System.out.println("Animal is eating");
   }
}
class Dog extends Animal {
   void bark() {
      System.out.println("Dog is barking");
   }
}
class Cat extends Animal {
   void meow() {
      System.out.println("Cat is meowing");
   }
}
```

In this example, both Dog and Cat are subclasses of Animal. They inherit the eat() method from Animal, and add their own specific methods bark() and meow().

12. Command Line Arguments in java:-

In Java, command line arguments are the parameters or inputs that are passed to the main method of a Java program when it is run from the command line. These arguments can be used to customize the behavior of the program based on user input.

Command line arguments are passed to the main method of a Java program as an array of strings, where each element of the array represents a separate argument. The first element of the array (args[0]) is the first argument, the second element (args[1]) is the second argument, and so on.

Here's an example of how to use command line arguments in Java:

```
public class CommandLineArgsExample {
   public static void main(String[] args) {
```

```
        if (args.length == 0) {
            System.out.println("No arguments provided");
        } else {
            System.out.println("Arguments:");
            for (int i = 0; i < args.length; i++) {
                System.out.println(args[i]);
            }
        }
    }
}
```

In this example, the program checks whether any command line arguments were provided. If no arguments were provided, it prints a message saying so. If arguments were provided, it prints each argument to the console.

For example, if we run the program with the following command line arguments:

java CommandLineArgsExample arg1 arg2 arg3
The output will be:
Arguments:
arg1
arg2
arg3

Method Overloading: It is the feature of Java that allows a class to have multiple methods with the same name, but with different parameters or arguments. The method that gets called during runtime depends on the number, order, and type of arguments passed. Here's an example:

```
public class Calculator {
  public int add(int a, int b) {
    return a + b;
  }
  public double add(double a, double b) {
    return a + b;
  }
  public int add(int a, int b, int c) {
    return a + b + c;
  }
```

```
}
```
In the above example, the add() method is overloaded with three different signatures. The first one takes two integers, the second one takes two doubles, and the third one takes three integers.

Method Overriding: It is the feature of Java that allows a subclass to provide its own implementation of a method that is already defined in its superclass. The method in the subclass must have the same name, return type, and parameters as the method in the superclass. Here's an example:

```
public class Animal {
  public void speak() {
    System.out.println("Animal speaks");
  }
}
public class Dog extends Animal {
  public void speak() {
    System.out.println("Dog barks");
  }
}
```
In the above example, the Dog class overrides the speak() method of the Animal class to provide its own implementation.

13. Final Keyword: It is a keyword in Java that is used to restrict the modification of variables, methods, and classes. If a variable is declared as final, its value cannot be changed. If a method is declared as final, it cannot be overridden by a subclass. If a class is declared as final, it cannot be subclassed. Here's an example:

```
public class Circle {
  public final double PI = 3.14159;
  public final void printArea() {
    System.out.println("Area is calculated using PI");
  }
}
```
In the above example, the PI variable is declared as final, which means its value cannot be changed. The printArea() method is also declared as final, which means it cannot be overridden by a subclass.

14. Super Keyword: It is a keyword in Java that is used to refer to the superclass of a subclass. It is used to call the constructor, methods, and variables of the superclass. Here's an example:

```
public class Animal {
  String name;
  public Animal(String name) {
    this.name = name;
  }
  public void speak() {
    System.out.println("Animal speaks");
  }
}
public class Dog extends Animal {
  public Dog(String name) {
    super(name);
  }
  public void speak() {
    System.out.println("Dog barks");
  }
}
```

In the above example, the Dog class calls the constructor of the Animal class using the super keyword to initialize the name variable.

15. Constructor with Super Keyword: It is a feature of Java that allows a subclass constructor to call the constructor of its superclass using the super keyword. This is useful when the superclass constructor needs to be called to initialize the instance variables of the subclass. Here's an example:

```
public class Animal {
  String name;
  public Animal(String name) {
    this.name = name;
  }
}
public class Dog extends Animal {
  String breed;
  public Dog(String name, String breed) {
    super(name);
    this.breed = breed;
  }
```

}
In the above example, the Dog class calls the constructor of the Animal class using the `super

16. JVM (Java Virtual Machine) is an abstract machine that executes Java bytecode. It is responsible for providing a runtime environment for Java applications. The JVM architecture consists of several components, as follows:

Class Loader: Class Loader is responsible for loading classes into the JVM. It loads the bytecode of the class and creates an internal representation of the class in the JVM.

Memory Area: Memory Area is a collection of various memory pools that are used by the JVM during the execution of a Java application. It is divided into three parts:

a. Heap Memory: Heap memory is the memory used for object allocation. It is shared among all threads of the application.

b. Stack Memory: Stack memory is used for storing method-specific data. Each method call creates a new frame in the stack memory.

c. Method Area: Method Area is used for storing class-level data, such as the bytecode of the methods, constant pool, and static fields.

Execution Engine: Execution Engine is responsible for executing the bytecode of the application. It consists of two parts:

a. Interpreter: Interpreter interprets the bytecode and executes the instructions one by one.

b. Just-In-Time Compiler (JIT): JIT compiles the frequently used bytecode into native machine code for faster execution.

Native Method Interface (NMI): Native Method Interface is used to call the native methods of the underlying operating system.

Example:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}
```

When we compile and run this code, the JVM executes the following steps:

Class Loader loads the bytecode of the HelloWorld class into the JVM.
Memory Area allocates memory for the HelloWorld object in the heap memory and memory for the main() method in the stack memory.
Execution Engine executes the bytecode of the main() method using the interpreter.
Interpreter executes the System.out.println("Hello, World!") statement, which prints "Hello, World!" to the console.
Native Method Interface is used to call the native methods of the operating system for printing to the console.
This is a basic example of how the JVM architecture works.

17. Inner class in java:-

In Java, an inner class is a class that is defined inside another class. It is used to encapsulate related functionality within a class and can access private members of the outer class. There are four types of inner classes in Java:

Nested inner class: A nested inner class is a class defined inside another class. It can access all members of the outer class, including private members. Here is an example:

```
public class OuterClass {
   private int x = 10;
   public class InnerClass {
      public void printX() {
         System.out.println("x = " + x);
      }
   }
}
// Create an instance of the outer class
OuterClass outer = new OuterClass();
```

```
// Create an instance of the inner class
OuterClass.InnerClass inner = outer.new InnerClass();
// Access a member of the inner class
inner.printX();
```

Static inner class: A static inner class is a nested class that is marked as static. It does not have access to the instance variables of the outer class. Here is an example:

```
public class OuterClass {
   private static int x = 10;
   public static class InnerClass {
      public void printX() {
         System.out.println("x = " + x);
      }
   }
}
// Create an instance of the inner class
OuterClass.InnerClass inner = new OuterClass.InnerClass();
// Access a member of the inner class
inner.printX();
```

Local inner class: A local inner class is a class defined inside a method or block. It has access to the variables of the enclosing method or block, but those variables must be final or effectively final. Here is an example:

```
public class OuterClass {
   public void printMessage(String message) {
      // Define a local inner class
      class InnerClass {
         public void print() {
            System.out.println(message);
         }
      }
      // Create an instance of the inner class
      InnerClass inner = new InnerClass();
      // Call a method of the inner class
      inner.print();
   }
}
// Create an instance of the outer class
OuterClass outer = new OuterClass();
```

```
// Call a method of the outer class
outer.printMessage("Hello, world!");
```

Anonymous inner class: An anonymous inner class is a class defined without a name. It is typically used to implement an interface or extend a class. Here is an example

```
public class OuterClass {
   public void printMessage(final String message) {
      // Create an instance of an anonymous inner class that implements Runnable
      new Runnable() {
         public void run() {
            System.out.println(message);
         }
      }.run();
   }
}
// Create an instance of the outer class
OuterClass outer = new OuterClass();
// Call a method of the outer class
outer.printMessage("Hello, world!");
```

18. Package:- In Java, a package is a way to organize related classes and interfaces into a single unit or namespace. It helps to avoid naming conflicts between classes and interfaces that have the same name and are used in different parts of a program. A package can contain other packages, classes, interfaces, and even other resources such as images and configuration files.

19. In Java, access control mechanism refers to the way in which access to class members (fields, methods, and nested classes) is regulated. Access control determines whether a class member can be accessed by code in another class or package.

Java provides four access control levels for class members:

private: Accessible only within the same class.
default or package-private: Accessible within the same package.
protected: Accessible within the same package and in subclasses.
public: Accessible from anywhere.
To specify an access control level for a class member, you use one of the access modifiers (private, default, protected, or public) before the member's declaration.

Here's an example:

```
public class Person {
   private String name; // accessible only within the Person class
   String address; // accessible within the same package
   protected int age; // accessible within the same package and in subclasses
   public String occupation; // accessible from anywhere

   // constructor and methods omitted for brevity
}
```

In this example, the name field is declared as private, so it can only be accessed from within the Person class. The address field is declared with default access, so it can be accessed from within the same package. The age field is declared as protected, so it can be accessed from within the same package and in subclasses. The occupation field is declared as public, so it can be accessed from anywhere.

By using access control, you can ensure that your class members are accessed only by the code that needs them and that you don't accidentally expose sensitive information to other parts of your program.

20. Interface in java: In Java, an interface is a collection of abstract methods (methods without implementation) and constant fields. An interface can be thought of as a contract that a class can choose to implement. By implementing an interface, a class agrees to implement all the methods defined in the interface.

Here's an example of a simple interface in Java:

```
public interface Shape {
  double getArea();
  double getPerimeter();
}

public class Rectangle implements Shape {
  private double length;
  private double width;

  public Rectangle(double length, double width) {
    this.length = length;
    this.width = width;
  }

  public double getArea() {
    return length * width;
  }

  public double getPerimeter() {
    return 2 * (length + width);
  }
}

public class Square implements Shape, Drawable {
  // implementation for Shape and Drawable methods
}
```

21. Dynamic Method lookup:

In Java, dynamic method lookup refers to the process of determining which implementation of a method to call at runtime, based on the actual type of the object that the method is called on. This allows for polymorphic behavior, where different objects of different types can respond to the same method call in different ways.

```
public class Animal {
  public void makeSound() {
    System.out.println("Some sound");
  }
```

```
}

public class Dog extends Animal {
  public void makeSound() {
    System.out.println("Bark");
  }
}

public class Cat extends Animal {
  public void makeSound() {
    System.out.println("Meow");
  }
}

public class Main {
  public static void main(String[] args) {
    Animal animal1 = new Dog();
    Animal animal2 = new Cat();

    animal1.makeSound(); // output: "Bark"
    animal2.makeSound(); // output: "Meow"
  }
}
```

In this example, we have defined a class hierarchy where Animal is the base class and Dog and Cat are derived classes. Each class has its own implementation of the makeSound() method.

In the Main class, we create two objects: animal1 of type Dog and animal2 of type Cat. Even though both objects are declared as type Animal, at runtime, the JVM determines the actual type of each object and uses the corresponding implementation of the makeSound() method.

When we call animal1.makeSound(), the JVM determines that animal1 is an instance of Dog, so it uses the Dog class's implementation of the makeSound() method, which outputs "Bark". Similarly, when we call animal2.makeSound(), the JVM determines that animal2 is an instance of Cat, so it uses the Cat class's implementation of the makeSound() method, which outputs "Meow".

This is an example of dynamic method lookup because the method implementation is determined at runtime based on the actual type of the object, rather than at compile-time based on the declared type of the variable.

22. Exception :

Java exception handling mechanism is a way to handle unexpected or exceptional situations that may occur during the execution of a Java program. Exceptions in Java are objects that represent an abnormal condition that has occurred during program execution. Java provides built-in exception classes for many common errors that can occur during program execution, as well as a way to create user-defined exceptions.

Java Exception Types:
There are two types of exceptions in Java: checked and unchecked exceptions.

Checked exceptions are those exceptions that are checked at compile-time by the Java compiler. These exceptions are also known as "compile-time exceptions" and are typically related to input/output operations, such as reading from a file or network socket. Checked exceptions must be handled or declared in the method signature using the "throws" keyword.

Unchecked exceptions are those exceptions that are not checked at compile-time by the Java compiler. These exceptions are also known as "runtime exceptions" and are typically related to programming errors, such as division by zero or accessing a null reference. Unchecked exceptions do not need to be handled or declared in the method signature.

Java Exception Handling Keywords:
Java provides several keywords for exception handling, including try, catch, throw, throws, and finally.

The "try" block is used to enclose the code that may generate an exception. The "catch" block is used to handle the exception that was thrown in the "try" block. The "throw" keyword is used to explicitly

throw an exception from a method. The "throws" keyword is used to declare the exceptions that a method may throw. The "finally" block is used to enclose the code that is always executed, regardless of whether an exception was thrown or not.

Built-in Exceptions:
Java provides a large number of built-in exception classes to handle many common errors that can occur during program execution. Some of the most common built-in exceptions are:

ArithmeticException: Thrown when an arithmetic operation has an exceptional condition, such as division by zero.
NullPointerException: Thrown when a null reference is encountered where an object is required.
ArrayIndexOutOfBoundsException: Thrown when an array index is out of bounds.
IOException: Thrown when an input/output operation fails, such as when reading from or writing to a file.
ClassNotFoundException: Thrown when a class cannot be found by the class loader.
User-Defined Exceptions:
Java also allows programmers to create their own exceptions by extending the built-in Exception class or one of its subclasses. By creating custom exceptions, programmers can define their own exceptional conditions that can be thrown and handled in a specific way.

Here's an example of a user-defined exception:

```
public class InvalidAgeException extends Exception {
  public InvalidAgeException(String message) {
    super(message);
  }
}

public class Person {
  private String name;
  private int age;

  public Person(String name, int age) throws InvalidAgeException {
    if (age < 0 || age > 120) {
```

```
    throw new InvalidAgeException("Invalid age: " + age);
   }
   this.name = name;
   this.age = age;
  }
}

public static void main(String[] args) {
  try {
   Person p = new Person("John", 150);
  } catch (InvalidAgeException e) {
   System.out.println(e.getMessage)
   }
}
```

23. String:

In Java, a String is an object that represents a sequence of characters. Java strings are immutable, meaning that once a string is created, its value cannot be changed. String objects can be created using the String constructor, which takes a sequence of characters as an argument.

Here is an example of creating a string using the String constructor:

```
String str = new String("Hello, world!");
```

String Operations:
Java provides a number of operations that can be performed on strings, including string extractions, string comparison, searching strings, modifying a string, and converting a string to other data types.

String Extraction:
To extract a substring from a string, we can use the substring() method. For example, to extract the first three characters of a string, we can do the following:

```
String str = "Hello, world!";
String sub = str.substring(0, 3); // sub is "Hel"
```

String Comparison:
To compare two strings, we can use the equals() or equalsIgnoreCase() method. For example:

```
String str1 = "Hello";
String str2 = "hello";
boolean result = str1.equalsIgnoreCase(str2); // result is true
```

Searching Strings:
To search for a substring within a string, we can use the indexOf() or lastIndexOf() method. For example:

```
String str = "Hello, world!";
int index = str.indexOf("world"); // index is 7
```

Modifying a String:
Since strings are immutable, we cannot modify their values directly. However, we can create a new string with the modified value using methods like concat(), replace(), or toUpperCase().

```
String str = "Hello, world!";
String newStr = str.replace("world", "Java"); // newStr is "Hello, Java!"
```

ToString() and valueOf() methods:
The toString() method is used to convert an object to a string. The valueOf() method is used to convert a primitive type to a string.

```
int num = 42;
String str = String.valueOf(num); // str is "42"
```

String Buffer:
In Java, a StringBuffer is a mutable sequence of characters that can be modified without creating a new object. A StringBuffer object is created using the StringBuffer constructor.

Here is an example of creating a StringBuffer object:

```
StringBuffer sb = new StringBuffer("Hello, world!");
```

StringBuffer Operations:
Java provides a number of operations that can be performed on StringBuffer objects, including appending, inserting, deleting, and replacing characters.

Appending:
To append a string to a StringBuffer object, we can use the append() method. For example:
StringBuffer sb = new StringBuffer("Hello");
sb.append(", world!"); // sb is now "Hello, world!"

Inserting:
To insert a string at a specific position in a StringBuffer object, we can use the insert() method. For example:
StringBuffer sb = new StringBuffer("Hello, world!");
sb.insert(7, "Java "); // sb is now "Hello, Java world!"

Deleting:
To delete a portion of a StringBuffer object, we can use the delete() method. For example:
StringBuffer sb = new StringBuffer("Hello, world!");
sb.delete(7, 12); // sb is now "Hello!"
Replacing:
To replace a portion of a StringBuffer object with another string, we can use the `replace

String Builder:
In Java, a StringBuilder is a mutable sequence of characters that can be modified without creating a new object. It is similar to the StringBuffer class but is not synchronized, which makes it more efficient in single-threaded environments.

StringBuilder provides a number of operations that can be performed on the underlying character sequence, including appending, inserting, deleting, and replacing characters.

Here is an example of creating a StringBuilder object:

StringBuilder sb = new StringBuilder("Hello, world!");
StringBuilder Operations:

Java provides a number of operations that can be performed on StringBuilder objects, including appending, inserting, deleting, and replacing characters.

Appending:
To append a string to a StringBuilder object, we can use the append() method. For example:
StringBuilder sb = new StringBuilder("Hello");
sb.append(", world!"); // sb is now "Hello, world!"

Inserting:
To insert a string at a specific position in a StringBuilder object, we can use the insert() method. For example:
StringBuilder sb = new StringBuilder("Hello, world!");
sb.insert(7, "Java "); // sb is now "Hello, Java world!"

Deleting:
To delete a portion of a StringBuilder object, we can use the delete() method. For example:
StringBuilder sb = new StringBuilder("Hello, world!");
sb.delete(7, 12); // sb is now "Hello!"

Replacing:
To replace a portion of a StringBuilder object with another string, we can use the replace() method. For example:
StringBuilder sb = new StringBuilder("Hello, world!");
sb.replace(7, 12, "Java"); // sb is now "Hello, Java!"

StringBuilder provides similar functionality to StringBuffer but is preferred in cases where synchronization is not required, as it is more efficient. However, if multiple threads will be modifying the same StringBuilder object, StringBuffer should be used instead to ensure thread safety.

~~24. I/O Stream:~~

~~I/O basics:~~
~~I/O stands for Input/Output. In Java, I/O is performed through streams, which are a sequence of bytes. Java provides two types of streams: Input~~

streams and Output streams. Input streams are used to read data from a source, such as a file or network connection, and output streams are used to write data to a destination, such as a file or network connection.

Streams:
In Java, there are two types of streams: byte streams and character streams.

Byte Streams: Byte streams are used to handle input and output of bytes. They are represented by the InputStream and OutputStream classes. Byte streams are used for reading and writing binary data, such as image or audio files.

Character Streams: Character streams are used to handle input and output of characters. They are represented by the Reader and Writer classes. Character streams are used for reading and writing text data, such as a text file.

Reading console input and writing console output:
To read data from the console in Java, we can use the Scanner class. Here is an example:

```
import java.util.Scanner;
public class ConsoleInput {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();
        System.out.println("Hello, " + name + "!");
    }
}
```

To write data to the console in Java, we can use the System.out.println() method. Here is an example:

```
public class ConsoleOutput {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
    }
```

Reading and writing files:

To read data from a file in Java, we can use the FileInputStream class. Here is an example:

```
import java.io.FileInputStream;
import java.io.IOException;
public class ReadFile {
  public static void main(String[] args) {
    try {
      FileInputStream fileInput = new FileInputStream("filename.txt");
      int data = fileInput.read();
      while (data != -1) {
        System.out.print((char) data);
        data = fileInput.read();
      }
      fileInput.close();
    } catch (IOException e) {
      System.out.println("An error occurred: " + e.getMessage());
    }
  }
}
```

To write data to a file in Java, we can use the FileOutputStream class. Here is an example:

```
import java.io.FileOutputStream;
import java.io.IOException;
public class WriteFile {
  public static void main(String[] args) {
    try {
      FileOutputStream fileOutput = new
FileOutputStream("filename.txt");
      String message = "Hello, world!";
      byte[] data = message.getBytes();
      fileOutput.write(data);
      fileOutput.close();
    } catch (IOException e) {
      System.out.println("An error occurred: " + e.getMessage());
    }
  }
}
```

25. Applets:

Applet class, architecture, and its skeleton:
In Java, an applet is a small program that runs within a web browser. It is based on the Applet class, which is a subclass of the Panel class. The architecture of an applet consists of three parts: the HTML page that contains the applet, the applet itself, and the web browser that displays the applet.

The skeleton code of an applet class in Java looks like this:

```
import java.applet.Applet;
import java.awt.Graphics;

public class MyApplet extends Applet {
    public void init() {
        // Initialization code here
    }

    public void start() {
        // Start code here
    }

    public void stop() {
        // Stop code here
    }

    public void destroy() {
        // Destruction code here
    }

    public void paint(Graphics g) {
        // Painting code here
    }
}
```

Applet life cycle methods, setForeground() and setBackground() methods:
The applet life cycle methods are used to manage the applet's state. They are as follows:

init(): This method is called when the applet is initialized, which happens only once when the applet is first loaded into memory.
start(): This method is called when the applet is started, which happens every time the applet is displayed on the screen.
stop(): This method is called when the applet is stopped, which happens when the applet is no longer visible on the screen.
destroy(): This method is called when the applet is destroyed, which happens when the web browser is closed or the applet is removed from the HTML page.
The setForeground() and setBackground() methods are used to set the foreground and background colors of the applet, respectively. Here is an example:

```
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;

public class ColorApplet extends Applet {
   public void init() {
      setBackground(Color.YELLOW);
   }

   public void paint(Graphics g) {
      g.setColor(Color.RED);
      g.fillRect(10, 10, 50, 50);
      g.setColor(Color.GREEN);
      g.fillRect(70, 10, 50, 50);
      g.setColor(Color.BLUE);
      g.fillRect(130, 10, 50, 50);
   }
}
```

Using the status window:
The status window is a small window that appears at the bottom of the web browser when an applet is running. It can be used to display messages to the user. To display a message in the status window, we can use the showStatus() method of the Applet class.
Here is an example:

```
import java.applet.Applet;
public class StatusApplet extends Applet {
   public void start() {
      showStatus("Applet started");
   }

   public void stop() {
      showStatus("Applet stopped");
   }
}
```

HTML APPLET tag, passing parameters to applet, getDocumentBase() and getCodeBase() methods:
The HTML APPLET tag is used to embed an applet in a web page. It has several attributes, such as code, width, height, and archive, that specify the applet's properties.
Here is an example:
<applet code="MyApplet.class" width="300" height="200"></applet>
We can pass parameters to the applet using the PARAM tag, which is a child element of the APPLET tag.

26. JDBC:-

Types of Drivers:
In JDBC, there are four types of drivers:

Type 1: JDBC-ODBC bridge driver
Type 2: Native-API/partly Java driver
Type 3: Net-protocol/all-Java driver
Type 4: Native-protocol/all-Java driver

JDBC Architecture:
JDBC (Java Database Connectivity) is a standard Java API for accessing databases. The JDBC architecture consists of two layers: the JDBC API and the JDBC driver API. The JDBC API provides a set of interfaces and classes for accessing databases from Java programs, while the JDBC driver API provides a standard interface for connecting to different types of databases.

JDBC classes and interfaces:
Some of the important classes and interfaces in JDBC are:

DriverManager: A class that manages the JDBC drivers.
Connection: An interface that represents a connection to a database.
Statement: An interface that represents a SQL statement.
ResultSet: An interface that represents the result of a SQL query.
PreparedStatement: An interface that represents a precompiled SQL statement.

Basic steps in developing JDBC applications:
The basic steps in developing JDBC applications are as follows:

Load the JDBC driver using the Class.forName() method.
Establish a connection to the database using the DriverManager.getConnection() method.
Create a statement object using the Connection.createStatement() method.
Execute a SQL query or update using the Statement.executeQuery() or Statement.executeUpdate() method.
Process the result set using the ResultSet object.
Close the result set, statement, and connection using the close() method.
Here is an example:

```
import java.sql.*;

public class JdbcDemo {
    public static void main(String[] args) throws SQLException {
        String url = "jdbc:mysql://localhost:3306/mydatabase";
        String username = "root";
        String password = "mypassword";

        // Load the JDBC driver
        Class.forName("com.mysql.jdbc.Driver");

        // Establish a connection to the database
        Connection conn = DriverManager.getConnection(url, username, password);

        // Create a statement object
        Statement stmt = conn.createStatement();
```

```
        // Execute a SQL query
        ResultSet rs = stmt.executeQuery("SELECT * FROM mytable");

        // Process the result set
        while (rs.next()) {
            String name = rs.getString("name");
            int age = rs.getInt("age");
            System.out.println("Name: " + name + ", Age: " + age);
        }

        // Close the result set, statement, and connection
        rs.close();
        stmt.close();
        conn.close();
    }
}
```

Creating a new database and table with JDBC:

To create a new database and table using JDBC, we can execute SQL statements using the Statement.executeUpdate() method. Here is an example:

```
import java.sql.*;

public class JdbcDemo {
    public static void main(String[] args) throws SQLException {
        String url = "jdbc:mysql://localhost:3306/";
        String username = "root";
        String password = "mypassword";

        // Load the JDBC driver
        Class.forName("com.mysql.jdbc.Driver");

        // Establish a connection to the database
        Connection conn = DriverManager.getConnection(url, username, password);

        // Create a new database
        Statement stmt = conn.createStatement();
        stmt.executeUpdate("CREATE DATABASE mydatabase");
```

```
        // Use the new database
        stmt.executeUpdate("USE mydatabase");

        // Create a new table
        stmt.executeUpdate("CREATE TABLE mytable (name VARCHAR(50),
age INT)");

        // Close the statement and connection
        stmt.close();
```

27. Delegation event model is a widely used event handling mechanism in Java programming. It allows the user to write code that is executed in response to certain actions, such as a button click, a mouse movement, or a keypress.

In the delegation event model, events are generated by various components, such as buttons, text fields, and checkboxes. These events are then passed to one or more listeners, which are registered with the components that generate the events.

Listeners can be implemented as separate classes, or they can be anonymous inner classes. When an event is generated, the component that generated the event notifies all of its listeners by calling a specific method in the listener's interface. The listener then responds to the event by executing the code contained in the listener method.

Let's take a simple example of a button click event to understand the delegation event model in Java. We can create a button and add an ActionListener to it, which will be notified whenever the button is clicked:

```
import java.awt.*;
import java.awt.event.*;

public class ButtonExample extends Frame {
   private Button button;

   public ButtonExample() {
      setTitle("Button Example");
```

```
        setSize(300, 200);

        button = new Button("Click Me");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button clicked!");
            }
        });

        setLayout(new FlowLayout());
        add(button);

        setVisible(true);
    }

    public static void main(String[] args) {
        new ButtonExample();
    }
}
```

In this example, we create a new button and add an anonymous ActionListener to it. When the button is clicked, the actionPerformed method is called, which simply prints a message to the console.

The delegation event model provides a flexible and powerful mechanism for handling events in Java applications. By using listeners, developers can write code that is executed in response to a wide variety of user actions, making their applications more interactive and engaging.

Q) Write Short notes on the following wrt java.
(a) Id and class selector
(b) Inner & Nested class
(c) ArrayList class
(d) for each statement

(a) Id and class selector: In Java, the id selector and class selector are used to select HTML elements based on their id or class attribute. The id selector is used to select an element with a specific id, while the class selector is used to select elements with a specific class.

For example, to select an element with the id "myDiv", we can use the following code:

Element element = document.getElementById("myDiv");
To select all elements with the class "myClass", we can use the following code:
Elements elements = document.getElementsByClass("myClass");

(b) Inner & Nested class: In Java, an inner class is a class that is defined inside another class. Inner classes can access the variables and methods of the enclosing class, and can be used to encapsulate related functionality within the outer class.

A nested class is a class that is defined inside another class, but is not an inner class. Nested classes can be static or non-static, and can be used to organize related classes within the same package or to hide implementation details from the rest of the program.

(c) ArrayList class: In Java, the ArrayList class is a resizable array implementation of the List interface. It is similar to an array, but can be dynamically resized and can store objects of any type. Elements can be added or removed from the ArrayList, and it provides methods for accessing and manipulating the elements in the list.

For example, to create a new ArrayList and add elements to it, we can use the following code:

ArrayList<String> list = new ArrayList<String>();
list.add("apple");
list.add("banana");
list.add("orange");

(d) for each statement: The for each statement is a loop construct in Java that allows you to iterate over a collection of elements, such as an array or ArrayList, without using an index variable. It is also known as the enhanced for loop.

For example, to iterate over an array of integers and print each element, we can use the following code:

```
int[] numbers = {1, 2, 3, 4, 5};
for (int num : numbers) {
   System.out.println(num);
}
```

In this example, the for each loop iterates over each element in the array and assigns it to the variable "num", which is then printed to the console.

Q) Differentiate between instance variables and class variables in java.

In Java, instance variables and class variables are two types of variables that can be used in a class.

Instance variables, also known as non-static variables, are declared within a class but outside of any method or constructor. Each instance of the class has its own copy of instance variables, which means that each instance can have different values for these variables. Instance variables are accessed using the dot notation with an instance of the class.

For example:

```
public class Car {
   String make;
   String model;
   int year;
}
```

In this example, "make", "model", and "year" are instance variables of the Car class. Each instance of the Car class will have its own copy of these variables.

Class variables, also known as static variables, are declared within a class but outside of any method or constructor with the "static" keyword. Class variables are shared by all instances of the class, which means that any changes made to a class variable will be visible to all instances of the class. Class variables are accessed using the class name and dot notation.

For example:

```
public class Bank {
   static int interestRate;
```

}
In this example, "interestRate" is a class variable of the Bank class. All instances of the Bank class will share the same "interestRate" value, and changes made to "interestRate" will be visible to all instances of the class.

To summarize, instance variables are unique to each instance of a class and are accessed using the dot notation with an instance of the class, while class variables are shared by all instances of a class and are accessed using the class name and dot notation.

Q) What makes Java a platform-independent language?

Java is considered a platform-independent language because of the following reasons:

Bytecode: When Java code is compiled, it is converted into an intermediate code called bytecode. Bytecode is a highly optimized and platform-independent code that can be executed on any platform that has a Java Virtual Machine (JVM) installed.

JVM: The JVM is a virtual machine that is responsible for interpreting the bytecode and executing it on the underlying operating system. The JVM provides an abstraction layer between the Java program and the underlying hardware and operating system, making it possible for Java code to run on any platform that has a JVM installed.

Platform-specific libraries: Java provides a set of platform-specific libraries, which are implemented differently for each operating system. These libraries provide access to platform-specific features such as windowing systems, file systems, and network protocols. Java programs can use these libraries to interact with the underlying operating system while still remaining platform-independent.

Java API: Java provides a rich set of APIs (Application Programming Interfaces) that are platform-independent. The Java API provides a set of classes and interfaces that can be used to perform common programming tasks such as file I/O, network communication, and database access.

By using bytecode, the JVM, platform-specific libraries, and the Java API, Java programs can be developed on one platform and executed on any platform that has a JVM installed, making Java a platform-independent language.

Q)Explain the importance of JVM and JRE.
JVM (Java Virtual Machine) and JRE (Java Runtime Environment) are two important components of the Java platform. Here's why they are important:

JVM: The JVM is a crucial component of the Java platform. It provides an environment for Java code to run on any hardware and operating system without any modifications to the code. The JVM interprets the Java bytecode and executes it on the underlying hardware and operating system. The JVM provides several benefits such as memory management, security, and exception handling. The JVM is responsible for making Java a platform-independent language.

JRE: The JRE is a set of tools and libraries that are required to run Java applications. It includes the JVM, class libraries, and other components required to run Java applications. The JRE provides a runtime environment for Java applications, which means that Java applications can be executed on any platform that has the JRE installed. The JRE provides several benefits such as automatic memory management, dynamic class loading, and security.

In summary, the JVM and JRE are critical components of the Java platform. The JVM provides an environment for Java code to run on any hardware and operating system, while the JRE provides the tools and libraries required to run Java applications. Without the JVM and JRE, it would not be possible to run Java applications on multiple platforms without any modifications to the code.

Q) Differentiate between 'throw' and 'throws' keywords in java.

In Java, both 'throw' and 'throws' are keywords used in exception handling, but they have different meanings and usage.

'throw' keyword: The 'throw' keyword is used to explicitly throw an exception from a method or a block of code. When an exception is thrown using the 'throw' keyword, the program control is immediately transferred to the nearest catch block that can handle the thrown exception. The syntax for using the 'throw' keyword is as follows:
throw new ExceptionType("Error Message");

'throws' keyword: The 'throws' keyword is used in the method declaration to indicate that the method may throw one or more checked exceptions. Checked exceptions are the exceptions that must be handled by the calling method or propagated further up the call stack. When a method throws a checked exception, it must declare the exception using the 'throws' keyword. The syntax for using the 'throws' keyword is as follows:

```
public void methodName() throws ExceptionType {
  // Method code that may throw an exception
}
```

To summarize, the 'throw' keyword is used to explicitly throw an exception, while the 'throws' keyword is used to declare that a method may throw an exception. The 'throw' keyword is used within a method or a block of code, while the 'throws' keyword is used in the method declaration.

Q) Differentiate between Vector and ArrayList.
Both Vector and ArrayList are dynamic arrays in Java, but they have some differences.

Synchronization: Vector is synchronized, which means that it is thread-safe, while ArrayList is not synchronized. This makes Vector slower than ArrayList in most cases.

Performance: Since Vector is synchronized, it requires locking to be thread-safe, which can lead to decreased performance compared to ArrayList, which does not require any locking. Therefore, ArrayList is faster than Vector in most cases.

Growth rate: Vector and ArrayList both dynamically grow and shrink as required, but they have different growth rates. Vector grows by 100% (doubles) of its size when it runs out of space, while ArrayList grows by

50% (half) of its size. This means that Vector may waste more memory than ArrayList when it grows.

Enumeration: Vector has a built-in enumeration feature, which allows you to iterate through its elements. ArrayList does not have this feature, but you can use the iterator() method to achieve the same functionality.

In summary, the main differences between Vector and ArrayList are synchronization, performance, growth rate, and enumeration. Vector is synchronized and has a larger growth rate, but it is slower and may waste more memory than ArrayList. ArrayList is faster and more memory-efficient, but it is not thread-safe.

Q)what is final class? how is it helpful in java?

In Java, a final class is a class that cannot be extended or subclassed. When a class is declared as final, it means that it cannot be inherited by any other class.

Final classes are helpful in Java for several reasons:

Security: By making a class final, you can prevent others from subclassing it and potentially overriding its methods or fields. This can be useful in situations where security is a concern.

Performance: Since a final class cannot be subclassed, the compiler can optimize the code more aggressively, which can result in improved performance.

Design: A final class can be used to represent a concept that should not be extended, such as a mathematical constant or a utility class.

Maintenance: By making a class final, you can ensure that its behavior remains consistent across different versions of the software. This can make maintenance and debugging easier in the long run.

In summary, a final class is a class that cannot be extended, and it is helpful in Java for providing security, improving performance, enforcing design decisions, and simplifying maintenance.

Q)Define a stream in java. Differentiate between a byte oriented and a character oriented streams in java.

In Java, a stream is a sequence of data that can be read from or written to a source or destination. Streams are used to read input from the keyboard, file, network, or any other source, and to write output to the console, file, network, or any other destination.

There are two types of streams in Java:

Byte oriented streams: Byte streams are used to handle data in binary format, such as images, videos, and other non-text files. They are represented by classes that end with "InputStream" and "OutputStream", such as FileInputStream and FileOutputStream. Byte streams read and write data in the form of bytes.

Character oriented streams: Character streams are used to handle data in text format, such as plain text files, HTML files, and XML files. They are represented by classes that end with "Reader" and "Writer", such as FileReader and FileWriter. Character streams read and write data in the form of characters.

The main difference between byte oriented and character oriented streams is how they handle data. Byte streams work with binary data, which means they read and write bytes directly. Character streams work with text data, which means they read and write characters, and can also handle character encoding and decoding.

In summary, streams in Java are sequences of data that can be read from or written to a source or destination. Byte oriented streams are used for binary data, while character oriented streams are used for text data.

Q)Differentiate between for each loop & iterator in java.

Both the for-each loop and iterator in Java are used to iterate over a collection of elements, such as an array or an ArrayList. However, they differ in their syntax, usage, and capabilities.

The for-each loop is a shorthand for the traditional for loop and provides an easy way to iterate over the elements of an array or a collection. Its syntax is as follows:

```
for (datatype element : collection) {
   // loop body
}
```

In this loop, the "datatype" represents the data type of the elements in the collection, "element" represents a variable that holds each element of the collection in turn, and "collection" represents the array or collection being iterated over.

The for-each loop is easy to use, but it has some limitations. For example, it cannot be used to modify the elements of the collection, and it does not provide an index of the current element.

On the other hand, the iterator is an object that allows you to traverse a collection of elements one by one, and also provides methods for adding, removing, and modifying elements of the collection. Its syntax is as follows:

```
Iterator<datatype> iterator = collection.iterator();
while (iterator.hasNext()) {
   datatype element = iterator.next();
   // loop body
}
```

In this loop, "iterator" represents the iterator object, "datatype" represents the data type of the elements in the collection, and "element" represents the current element being iterated over.

The iterator provides more functionality than the for-each loop, such as the ability to modify the elements of the collection, and it also provides more control over the iteration process. However, it is more verbose and requires more code to use.

In summary, the for-each loop is a simpler and more concise way to iterate over a collection of elements, but it has some limitations. The iterator provides more functionality and control over the iteration process, but it requires more code to use.

Q)what is adapter class in java?

In Java, an adapter class is a class that provides default implementations of methods for an interface. It allows a class to implement only the methods it needs from an interface, while providing empty or default implementations for the rest of the methods.

An adapter class is useful when you want to create an object that implements an interface, but you do not want to provide an implementation for all the methods of the interface. By using an adapter class, you can provide an implementation for only the methods you need, and the adapter class will provide default implementations for the rest.

For example, the java.awt.event.MouseAdapter is an adapter class in Java. It provides empty implementations for all the methods of the MouseListener interface, MouseMotionListener interface, and MouseWheelListener interface. This allows a programmer to create a custom listener by extending the MouseAdapter class and implementing only the methods they need.
Here is an example of using the MouseAdapter class:

```
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
public class MyMouseListener extends MouseAdapter {
   @Override
   public void mouseClicked(MouseEvent e) {
      System.out.println("Mouse clicked at " + e.getX() + ", " + e.getY());
   }
}
// In main method or wherever the mouse listener needs to be added
MyMouseListener mouseListener = new MyMouseListener();
someComponent.addMouseListener(mouseListener);
```

In this example, we create a custom mouse listener called "MyMouseListener" by extending the MouseAdapter class and implementing only the "mouseClicked" method. We then add this listener to a component using the "addMouseListener" method. When the user clicks on the component, the "mouseClicked" method in the MyMouseListener class will be called and will print out the coordinates of the mouse click.

Q)what is the use of finalize() method in java?

In Java, the finalize() method is a method of the Object class that is called by the garbage collector when it determines that an object is no longer being used by the program and is eligible for garbage collection.

The finalize() method can be overridden by a class to perform some cleanup tasks before the object is garbage collected. For example, a class that opens a file or a database connection may use the finalize() method to close the file or connection before the object is garbage collected.

However, it is generally not recommended to use the finalize() method for resource cleanup as there is no guarantee when or if the method will be called by the garbage collector. Instead, it is better to use try-with-resources or a similar construct to ensure that resources are properly cleaned up when they are no longer needed.

Here is an example of how to override the finalize() method:

```
public class MyClass {
   private Connection connection;

   public MyClass() {
      // Open the database connection
      connection =
DriverManager.getConnection("jdbc:mysql://localhost/mydb",
"username", "password");
   }

   @Override
   protected void finalize() throws Throwable {
      // Close the database connection
      connection.close();
      super.finalize();
   }
}
```

In this example, the MyClass class opens a database connection in its constructor. The finalize() method is overridden to close the database connection before the object is garbage collected. Note that the super.finalize() method is called at the end of the method to ensure that the object is properly garbage collected.

Q)what is printwriter class in java?

In Java, the PrintWriter class is a character-oriented output stream that can be used to write formatted text to a file or other output destination. It is a convenient way to write text to a file because it supports various formatting options, such as printing integers and floating-point numbers in a specified format, and writing strings with a specified encoding.

Here is an example of how to use the PrintWriter class to write text to a file:

```
import java.io.*;

public class Example {
    public static void main(String[] args) throws IOException {
        PrintWriter writer = new PrintWriter("output.txt", "UTF-8");
        writer.println("Hello, world!");
        writer.printf("The value of pi is approximately %.2f", Math.PI);
        writer.close();
    }
}
```

In this example, a new PrintWriter object is created with the file name "output.txt" and the character encoding "UTF-8". The println() method is used to write the string "Hello, world!" to the file, and the printf() method is used to write the value of pi to the file with two decimal places. Finally, the close() method is called to close the file and flush any remaining data to the file.

Note that if the specified file does not exist, a new file will be created with that name. If the file already exists, its contents will be overwritten. Also, if an IOException occurs while writing to the file, the PrintWriter will automatically close the file and throw the exception. Therefore, it is important to handle any exceptions that may be thrown when using a PrintWriter.