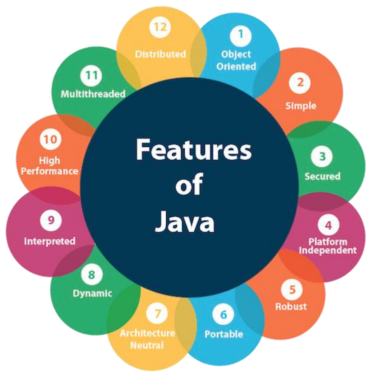
1.What is Java?

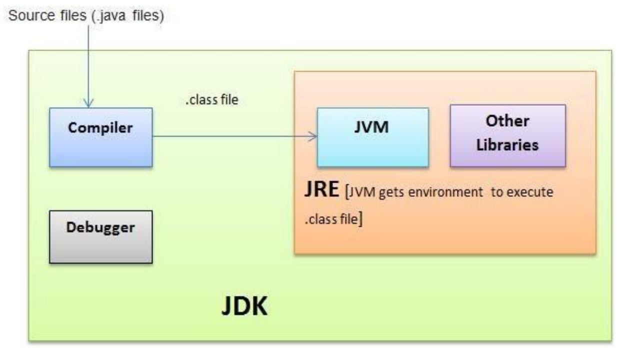
Java is a versatile, **platform-independent programming language** known for its object-oriented nature, security features, and robustness. It allows developers to write code once and run it on different platforms, making it widely used in various applications, including web development, Android app development, and server-side programming.

2.Key Features of Java:

1. **Platform Independence**: Java code can run on any platform (Windows, Mac, Linux, etc.) because it's compiled into an intermediate form called **bytecode**, which is executed by the Java Virtual Machine (JVM).
2. **Object-Oriented**: Java follows the principles of object-oriented programming (OOP), emphasizing classes, objects, inheritance, and polymorphism.
3. **Simple and Easy to Learn**: Java has a straightforward syntax, making it accessible for beginners. It avoids complex features found in other languages.
4. **Robust and Secure**: Java's strong type-checking, exception handling, and memory management contribute to its robustness. It also provides security features like sandboxing and access control.
5. **Multithreading and Concurrency**: Java supports multithreading, allowing programs to execute multiple tasks concurrently. This is crucial for performance and responsiveness.
6. **Rich Standard Library**: Java comes with a vast standard library that provides ready-to-use classes and methods for common tasks.
7. **Garbage Collection**: Java automatically manages memory by reclaiming unused objects through garbage collection.
8. **High Performance**: Although interpreted, Java's Just-In-Time (JIT) compilation optimizes performance during execution.



3. JDK, JRE and JVM


Source files (.java files)
.class file
Compiler
JVM
Other Libraries
JRE [JVM gets environment to execute .class file]
Debugger
JDK

1. **JDK (Java Development Kit)**:
   - The JDK is a software package that provides tools and utilities for **developing, compiling, and running Java applications**.
   - It includes the **Java compiler**, which translates human-readable Java code into bytecode.
   - Developers use the JDK to create and build Java applications.

2. **JVM (Java Virtual Machine)**:
   - The JVM is an **abstract computing machine** that executes Java bytecode.
   - It allows Java programs to be **platform-independent** by converting bytecode into machine-specific code.
   - JVM handles tasks like **loading code**, **verifying code**, **executing code**, and providing a **runtime environment**.
   - In essence, it bridges the gap between the Java language and the underlying hardware and operating systems.

3. **JRE (Java Runtime Environment)**:
   - The JRE includes the **JVM** and the **standard Java APIs (core classes and supporting files)**.
   - It provides just enough to **run Java applications**, but not enough to compile them.
   - When you want to **execute** a Java program, you need the JRE.

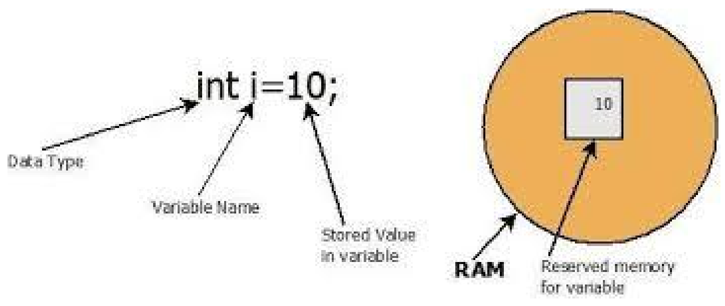
Remember, JDK is for development, JRE is for running applications, and JVM is the runtime environment!

Sl.No	JRE	JVM	JDK
1.	JRE is an acronym for Java Runtime Environment. It is also written as Java RTE.	JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides a runtime environment in which Java bytecode can be executed.	JDK (Java Development Kit) is a software development kit to develop applications in Java. In addition to JRE, JDK also contains number of development tools (compilers, JavaDoc, Java Debugger etc.).
2.	JRE is platform dependent.	JVM is platform independent.	JDK is platform dependent i.e for different platforms different JDK required.
3.	JRE contains class libraries and other supporting files that JVM requires to run the program.	JVM does not include software development tools.	It contains tools for developing, debugging and monitoring java application.
4.	JRE = Java Virtual Machine (JVM) + Libraries to run the application.	JVM = Only Runtime environment for executing the Java byte code.	JDK = Java Runtime Environment (JRE) + Development tools.

1. **Variables:**

- A **variable** is like a labeled box where you can put things.
- In programming, a variable is a **named memory location** that holds a value.
- Think of it as a container for data.
- You give it a name (like “number” or “name”), and it can store different types of
- information.
- For example:

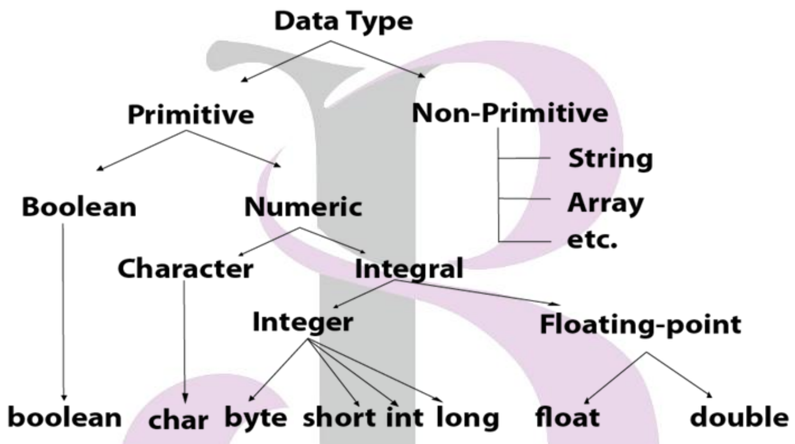
```
int number = 42; // Here, 'number' is a variable of type int
(integer).
boolean flag = true; // 'flag' is a variable of type boolean.
String name = "Alice"; // 'name' is a variable of type String.
```



2. Data Types:

The **data type** is like a tag on the box that tells you what kind of things the variable can hold.

- It specifies the type and quantity of value a variable can store.
- In Java, you must specify the data type before the variable name.
- Common data types include:
  - **int**: For whole numbers (e.g., 42).
  - **Boolean**: For true/false values.
  - **String**: For text (e.g., "Hello, world!")



4.WHAT IS BLOCK IN JAVA

In Java, a **block** is a group of zero or more statements enclosed within curly braces { }. These blocks serve various purposes in your code. Let's explore the different types of blocks:

1. **Method Blocks**:

- These blocks define a method's behavior and enclose a set of statements.
- Examples:

```
public static int addNumbers(int a, int b) {
int sum = a + b;
return sum;
}
```

2. **Conditional Blocks**:

- Enclosed by if, else if, and else statements, these blocks control code execution based on conditions.

- Example:

```
int number = 10;
if (number > 0) {
System.out.println("Number is positive.");
} else if (number < 0) {
System.out.println("Number is negative.");
} else {
System.out.println("Number is zero.");
}
```

3. **Loop Blocks**:
   - Created by while, for, and do-while loops, these blocks allow repeated execution of statements.
   - Example (using a for loop):

```
for (int i = 1; i <= 5; i++) {
System.out.println("Iteration " + i);
}
```

4. **Class Blocks**:
   - Define the scope of a class and contain field and method declarations.
   - Example:

```
public class MyClass {
int x; // Field declaration
public MyClass() {
x = 5; // Initialization within constructor block
}
public void displayX() {

System.out.println("Value of x: " + x);
}
}
```

5. **Initialization Blocks**:
   - Used to initialize fields or perform additional setup during instance creation.
   - Two types: instance and static initialization blocks.
   - Example (instance initialization block):

```
public class InitializationExample {
{
// Instance initialization block
// Initialize fields or perform setup
}
}
```

In Java, **static blocks** are used to initialize static variables or perform other setup tasks when a class is loaded into memory. Here's how they work:

1. **Static Initialization Blocks**:
   - Enclosed within static { ... }, these blocks execute once when the class
   - is loaded.
   - Useful for initializing static fields or performing other one-time setup.
   - Example:

```
public class MyClass {
static int count;
static {
count = 0; // Initialize static field
System.out.println("Static block executed.");
}
}
```

2. **Order of Execution**:
   - Static blocks execute in the order they appear in the code.
   - They run before any static methods or field initializations.
   - Example:

```
public class OrderExample {
static {
System.out.println("First static block");
}
public static void main(String[] args) {

System.out.println("Main method");
}
static {
System.out.println("Second static block");
}
```

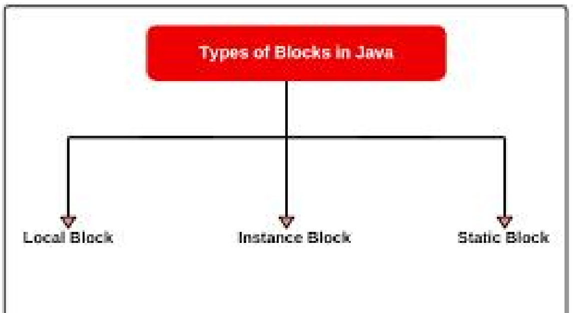
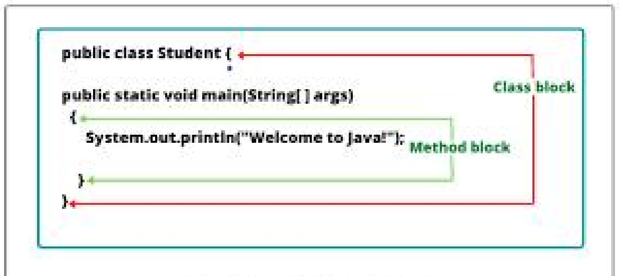


Fig: Types of blocks in Java



```
public class Student {

public static void main(String[ ] args)
 {
  System.out.println("Welcome to Java!");
 }
}
```

Fig: Class block and Method block in Java

}

5.WHAT ARE METHODS IN JAVA

1. **What Is a Method?**

- A method is a collection of statements grouped together to perform a specific
- task or operation.
- It encapsulates functionality and promotes code reusability.
- You write a method once and use it multiple times throughout your program.

2. **Method Declaration Components:**

- **Method Signature**: Includes the method name and parameter list.
- **Access Specifier**: Determines the visibility of the method (e.g., public, private, protected, or default).
- **Return Type**: Specifies the data type that the method returns (e.g., int, String,
- or void for no return value).
- **Method Name**: A unique name that reflects the method's purpose.
- **Parameter List**: Contains data type and variable names (if the method takes any
- arguments).
- **Method Body**: The actual code inside the method, enclosed in curly braces.

3. **Example of a Simple Method:**

```
public class Example {
public static void greetUser(String name) {
System.out.println("Hello, " + name + "!");
}
public static void main(String[] args) {
greetUser("Alice");
greetUser("Bob");
}
}
```

- In this example, the greetUser method takes a String argument (name) and prints a personalized greeting.

4. **Main Method (Entry Point):**

- The main method is the starting point for executing a Java program.
- It has the following signature:

```
public static void main(String[] args) {
// Your program logic goes here
}
```

5. **Types of Methods:**

- **Static Methods**: Belong to the class itself (not specific to instances).
- **Instance Methods**: Associated with an object (instance) of the class.
- **Constructor Methods**: Used for object initialization (same name as the class).
- **Getter and Setter Methods**: Access and modify class fields.
- **User-defined methods**: Custom methods created by developers.

6.What are the Access Specifiers in java

In Java, **access specifiers** control the visibility and accessibility of classes, methods, and fields. Let's explore the four types of access specifiers:

1. **Default (Package-Private)**:
   - When no access modifier is specified (i.e., no public, private, or protected),
   - it defaults to package-private.
   - Members (classes, methods, fields) with default access are accessible only
   - within the same package.
   - Example:

```
// Package p1
class MyClass {
void display() {
System.out.println("Package-private method");
}
}
```

2. **Private**:

- Declared using the private keyword.
- Private members are accessible only within the same class.
- Example:

```
class A {
private void secretMethod() {
System.out.println("Private method");
}
}
```

3. **Protected**:

- Specified with the protected keyword.
- Accessible within the same package and by subclasses (even in different packages).
- Example:

```
class B extends A {
void useProtectedMethod() {
secretMethod(); // Accessing protected method
}
}
```

4. **Public**:

- Denoted by the public keyword.
- Members are accessible from any class or package.
- Example:

```
public class Main {
public static void main(String[] args) {
MyClass obj = new MyClass();
obj.display(); // Accessing package-private method
}}
```

6. what is method overloading and its use in java

In Java, **method overloading** allows different methods to have the same name but different signatures. The signature can differ by the number or type of input parameters, or a combination of both. Here's how it works:

1. **Changing the Number of Parameters**:

- You can achieve method overloading by changing the number of parameters while passing them to different methods.
- Example:

```
public class Product {
public int multiply(int a, int b) {
return a * b;
}
public int multiply(int a, int b, int c) {
return a * b * c;
}
}
```

2. **Changing Data Types of the Arguments**:

- Methods can also be considered overloaded if they have the same name but different parameter types.
- Example:

```
public class Product {
public int prod(int a, int b, int c) {
return a * b * c;
}
public double prod(double a, double b, double c) {
return a * b * c;
}
}
```

3.Changing the Order of Parameters:

- Rearranging the parameters of two or more overloaded methods can also achieve method overloading
- For example, if one method has parameters (String name, int Roll no) and another has (int roll no, String name), both with the same name, they are considered overloaded.

8.WHAT IS CLASS AND OBJECT

1. **Class in Java**:
   - A **class** serves as a blueprint or template for creating **objects**. It defines the attributes (also known as data members) and behaviors (methods) that objects of a certain type will exhibit.
   - Think of a class as a high-level description of a real-world concept or entity. For example, if we're building a system to manage employees, we might have an Employee class.
   - Key points about classes:
     - **Attributes**: These represent the state of an object. For instance, an Employee class might have attributes like name, employeeId, and salary.
     - **Methods**: These define the behavior of an object. Methods allow us to perform actions related to the object. In our Employee class, methods could include calculateSalary(), promote(), and hire().
     - **Template**: A class is not a real-world entity; it's a blueprint. It doesn't occupy memory by itself.
     - **Example**:

```
public class Employee {
// Data members (attributes)
private String name;
private int employeeId;
private double salary;
// Methods
public void calculateSalary() {
// Implementation to calculate salary
}
public void promote() {
// Implementation for promotion logic
}
}
```

2. **Object in Java**:
   - An **object** is an instance of a class. It represents a specific real-world entity based on the class definition.o Objects have both state (attributes) and behavior (methods). When we create an object, we're essentially creating a tangible representation of the class.
   - For example, if we create an Employee object named john, it will have its own name, employeeId, and salary.
   - Key points about objects:

- **State**: Represents the current values of attributes. For john, the state includes his name, ID, and salary.
- **Behavior**: Objects can perform actions using methods. john can
- calculate his salary or get promoted.
- **Interaction**: Objects interact by invoking each other's methods.
- **Example**:

```
public class Main {
public static void main(String[] args) {
// Creating an Employee object
Employee john = new Employee();
john.name = "John Doe";
john.employeeId = 12345;
john.salary = 75000.0;
// Using methods
john.calculateSalary();
john.promote();
}
}
```

In summary, classes define the structure, while objects represent specific instances based on that structure.

9.What is constructor and constructor overloading:

1. **Constructor**:
   - A constructor is like a special method in Java, but it has no return type.
   - It runs automatically when you create an object of a class.
   - Think of it as the "setup" phase for an object. It initializes its state (attributes) or performs any necessary setup.
   - Constructors have the same name as the class they belong to.
   - Example:

```
class Person {
String name;
int age;
// Constructor
Person(String n, int a) {
name = n; age = a;
}
}
```

2. **Constructor Overloading**:
   - Imagine you want to create objects in different ways. Maybe you want to set different initial values for an object's attributes.
   - Constructor overloading allows you to define multiple constructors in the same class with different parameter lists.
   - Each constructor can perform a different task based on the provided arguments.
   - Example:

```
class Box {
double width, height, depth;
// Constructor with three arguments
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// Default constructor (no arguments)
Box() {
width = height = depth = 0;
}
// Constructor for a cube (one argument)
Box(double len) {
width = height = depth = len;
}
}
```

- Now you can create boxes like this:

```
Box myBox1 = new Box(10, 20, 15); // Custom dimensions
Box myBox2 = new Box(); // Default dimensions
Box myCube = new Box(7); // Cube with equal sides
```

10.What is new keyword in Java
The new keyword in Java is used to create an instance of a class. Essentially, it allocates memory for a new object and returns a reference to that memory. You can also use new to create array objects. Here are some examples:
1. Creating an object using new and invoking a method:

```
public class NewExample1 {
void display() {
System.out.println("Invoking Method");
}
public static void main(String[] args) {
NewExample1 obj = new NewExample1();
obj.display();
}
}
// Output: Invoking Method
```

11.what is this,super keywords in java

1. **this** Keyword:

- The this keyword refers to the **current instance** of the class. It is used within an instance method or constructor to refer to the object on which the method or constructor is invoked.
- Common uses of this:

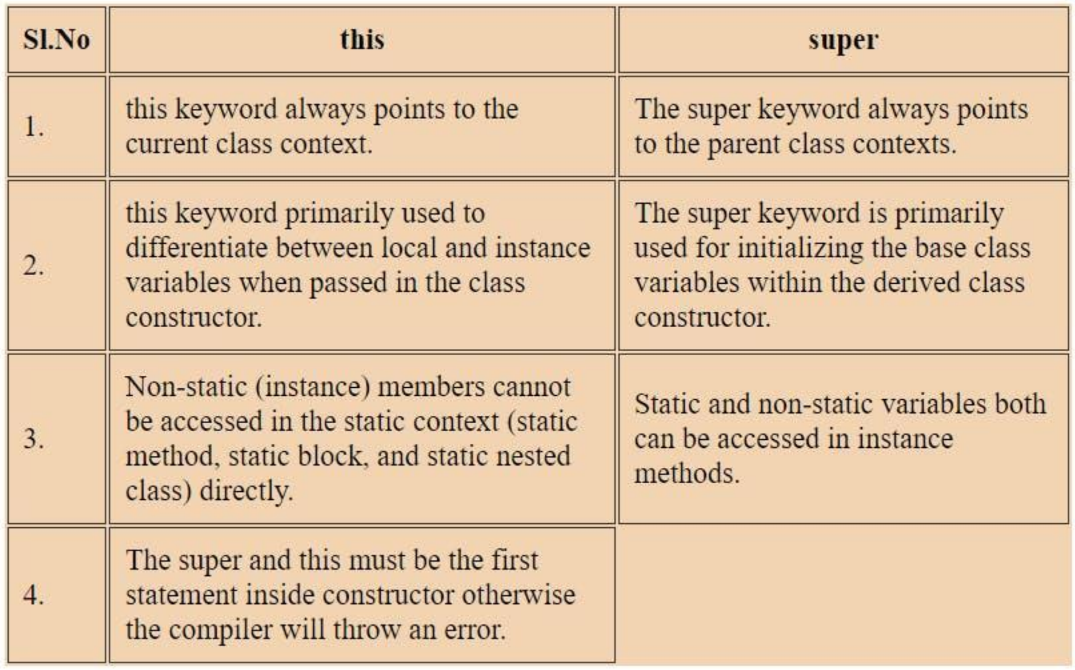
  - ➢ To differentiate between instance variables and method parameters with the same name.
  - ➢ To call another constructor within the same class (constructor chaining).
  - ➢ To return the current instance from a method (method chaining).
- Example:

```
class MyClass {
int value;
MyClass(int value) {
this.value = value; // Refers to the instance variable
}
void display() {
System.out.println("Value: " + this.value);
}
}
public class Main { public static void main(String[] args) {
MyClass obj = new MyClass(42);
obj.display();
}
}
// Output: Value: 42
```

2. **super** Keyword:

- The super keyword refers to the **parent class** (superclass) of the current subclass. It allows you to access members (fields, methods, and constructors) of the parent class.
- Common uses of super:
  - ➢ To call a superclass constructor from a subclass constructor (using super()).
  - ➢ To invoke a method defined in the parent class (even if overridden in the subclass).
  - ➢ To access fields defined in the parent class.
- Example:

```
class Vehicle {
int maxSpeed = 120;
}
class Car extends Vehicle {
int maxSpeed = 180;
void display() {
System.out.println("Maximum Speed: " + super.maxSpeed);
// Refers to the parent class field
}
}
public class Main {
public static void main(String[] args) {
Car smallCar = new Car();
smallCar.display();
}
}
// Output: Maximum Speed: 120
```



Sl.No	this	super
1.	this keyword always points to the current class context.	The super keyword always points to the parent class contexts.
2.	this keyword primarily used to differentiate between local and instance variables when passed in the class constructor.	The super keyword is primarily used for initializing the base class variables within the derived class constructor.
3.	Non-static (instance) members cannot be accessed in the static context (static method, static block, and static nested class) directly.	Static and non-static variables both can be accessed in instance methods.
4.	The super and this must be the first statement inside constructor otherwise the compiler will throw an error.	

13.what is inheritance, polymorphism, abstraction, encapsulation
Certainly! Let's explore the fundamental concepts of **Object-Oriented Programming (OOP)**:

1. **Inheritance**:

- **Definition**: Inheritance allows you to create class hierarchies, where a base class (parent class) provides its behavior and attributes to a derived class (subclass).
- **Purpose**: It enables code reuse and the creation of new abstractions based on existing ones.
- **Example**: If we have a Bird class with common properties like color, legs, and methods like eat() and fly(), we can create specific bird types (e.g., Pigeon) by inheriting from the Bird class. The derived class can override or extend the inherited functionality.
- **Example Code**:

```
class Bird {
String color;
int legs;
public void eat() {
System.out.println("This bird has eaten");
}
public void fly() {
System.out.println("This bird is flying");
}
}class Pigeon extends Bird {
public void fly() {
```

```
System.out.println("Pigeon flies!");
}
}
```

2. **Polymorphism**:
   - **Definition**: Polymorphism allows you to implement inherited properties or methods in different ways across multiple abstractions.
   - **Purpose**: It ensures that the proper method will be executed based on the calling object's type.
   - **Example**: When calling a method (e.g., fly()) on a bird object (e.g., Pigeon), the appropriate implementation (from the subclass) is invoked.
   - **Example Code**:

```
Bird pigeon = new Pigeon();
pigeon.fly(); // Calls Pigeon's fly() method
```

3. **Abstraction**:
   - **Definition**: Abstraction involves hiding the internal state and functionality of an object, allowing access only through a public set of functions (methods).
   - **Purpose**: It simplifies complex systems by focusing on essential features and ignoring implementation details.
   - **Example**: Encapsulating bird properties (like color) within methods (getters and setters) ensures controlled access and prevents direct modification.
   - **Example Code**:

```
class Bird {
private String color; // Encapsulated property
public void setColor(String newColor) {
this.color = newColor;
}
public String getColor() {
return color;
}
}
```

4. **Encapsulation**:
   - **Definition**: Encapsulation restricts direct access to an object's attributes and ensures that changes occur through well-defined methods.o **Purpose**: It enhances security, maintains consistency, and prevents unintended modifications.
   - **Example**: Using getters and setters to access and modify bird properties (e.g., color) while maintaining control over their values.
   - **Example Code**:

```
Bird pigeon = new Bird();
pigeon.setColor("Grey"); // Set color using setter
String pigeonColor = pigeon.getColor(); // Get color using
getter
```

14.what is constructor chaining and its rules
Constructor chaining in Java refers to the process of calling one constructor from another constructor within the same class or from a base class (superclass). It allows you to initialize an object by invoking multiple constructors in a sequence.
Let's explore how constructor chaining works:

1. **Within the Same Class (Using** this**)**:
   - In the same class, you can use the this() keyword to invoke another constructor.
   - The this() expression should always be the **first line** of the constructor.
   - Example:

```
class Temp {
Temp() {
this(5); // Calls the second constructor
System.out.println("The Default constructor");
}
Temp(int x) {
this(5, 15); // Calls the third constructor
System.out.println(x);
}
Temp(int x, int y) {
System.out.println(x * y);
}
public static void main(String args[]) {
new Temp();
}
}
// Output: 75
```

   - Rules:
     - ➢ There should be at least one constructor without the this() keyword (as shown in the third constructor above).
     - ➢ Constructor chaining can occur in any order.

2. **To Another Class (Using** super**)**:
   - When inheriting from a base class, you can use the super() keyword to call a constructor from the parent class.
   - The subclass constructor's task is to call the superclass constructor first.
   - Example:

```
class Base {
String name;
Base() {
this(""); // Calls the parameterized constructor of
Base
System.out.println("No-argument constructor of base
class");
}
Base(String name) {
this.name = name;
System.out.println("Calling parameterized constructor
of base");
}
}
class Derived extends Base {
Derived() {
System.out.println("No-argument constructor of derived
class");
}
Derived(String name) {
super(name); // Calls the parameterized constructor of
Base
System.out.println("Calling parameterized constructor
of derived");
}
public static void main(String args[]) {
Derived obj = new Derived("test");
}
}
// Output:
// Calling parameterized constructor of base
// No-argument constructor of derived class
// Calling parameterized constructor of derived
```

15.What is JAVA bean (getters an setters)
Definition: A JavaBean is a Java class that encapsulates multiple objects into a single standardized object (the bean). It adheres to the following conventions:

- Has a no-argument constructor.
- Is **Serializable** (can be saved to a file or transmitted over a network).
- **Purpose**:
  - **Reusability**: JavaBeans allow you to create components that can be reused across different parts of your application.
  - **Maintenance**: By encapsulating related objects, it simplifies maintenance and enhances code organization.

16.what is exception handling in java
Exception handling in Java is a powerful mechanism to manage runtime errors and maintain the normal flow of an application. Let's dive into the details:

1. **What is an Exception in Java?**

- An exception is an abnormal condition that disrupts the normal flow of a program.
- In Java, exceptions are objects that are thrown at runtime when an error occurs.

2. **What is Exception Handling?**

- Exception handling is a mechanism to deal with runtime errors, such as ClassNotFoundException, IOException, SQLException, and more.
- It ensures that even if an exception occurs, the program can continue executing without abruptly terminating.

```
statement1;
statement2;
statement3;
statement4;
statement5; // Exception occurs here
statement6; // Rest of the code will not execute
without exception handling
statement7;
statement8;
statement9;
statement10;
```

1. **Try-Catch Blocks**:
   - The try block is used to enclose code that might throw an exception.
   - If an exception occurs within the try block, the control transfers to the corresponding catch block.
   - The catch block catches the exception and executes statements specific to handling that exception.
2. **Finally Block**:
   - The finally block is optional and follows a try-catch block.
   - It is used for cleanup tasks (e.g., closing files, releasing resources) and always executes, regardless of whether an exception occurred or not.

17.What is StringBuffer:
a. A peer class of String.
b. Provides functionality similar to strings.
c. Represents a **fixed-length, immutable** character sequence.
d. Allows insertion of characters or substrings in the middle or appending to the end.
e. Automatically grows to accommodate additions.
f. Preallocates more characters than needed for growth.
g. Example:

```
StringBuffer str = new StringBuffer("Hello");
str.append(" World!");
System.out.println(str); // Output: Hello World!
```

18. What is StringBuilder:
a. Similar to StringBuffer.
b. Represents a **mutable** sequence of characters.
c. Preferred for single-threaded programs due to better performance.
d. Example:

```
StringBuilder str = new StringBuilder("Hello");
str.append(" World!");
System.out.println(str); // Output: Hello World!
```

Conversion:
e. To convert from StringBuffer to StringBuilder, first use toString() to get
a String object, then create a StringBuilder.

```
StringBuffer sbr = new StringBuffer("Geeks");
String str = sbr.toString();
StringBuilder sbl = new StringBuilder(str);
```

19.what is the difference between final,finally,finalize

1.final:
o **Definition**: final is a keyword used to declare entities (variables, methods, or classes) that cannot be modified after initialization.

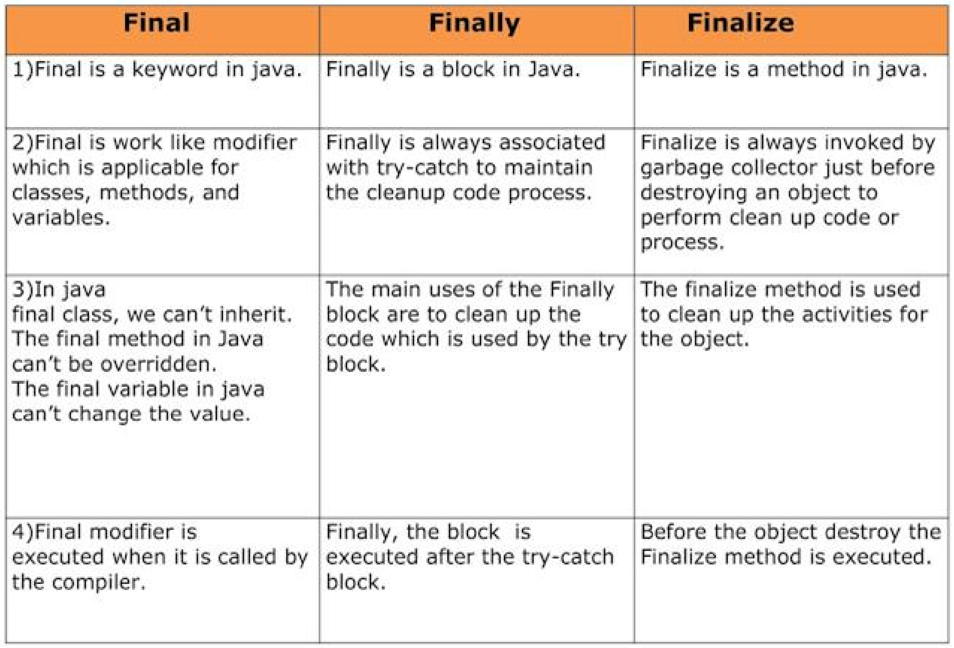
- **Applicability**:
  - ➢ Variables: Once declared, a final variable becomes constant and cannot be changed.
  - ➢ Methods: A final method cannot be overridden by a subclass.
  - ➢ Classes: A final class cannot be inherited.
- **Execution**: A final method is executed only when explicitly called.

2. finally:
- **Definition**: finally is a block used in exception handling.
- **Functionality**:
  - ➢ It runs important code whether an exception occurs or not.
  - ➢ Useful for cleanup (e.g., releasing resources) after a try block.
- **Execution**: The finally block executes as soon as the associated try catch block completes, regardless of exceptions.

3. finalize:
- **Definition**: finalize is a method from the Object class.
- **Functionality**:
  - ➢ Called by the garbage collector before an object's memory is reclaimed.
  - ➢ Used for resource cleanup (e.g., closing files, releasing network connections).
- **Caution**: Avoid using it directly; prefer other resource management techniques.
- Example:

```
class Example {
final int age = 18; // Final variable
void someMethod() {
```

```
// Code here
}
public static void main(String[] args) {
try {
// Code that may throw an exception
} catch (Exception e) {
// Exception handling
} finally {
// Cleanup code (always executed)
}
}
}
```



Final	Finally	Finalize
1)Final is a keyword in java.	Finally is a block in Java.	Finalize is a method in java.
2)Final is work like modifier which is applicable for classes, methods, and variables.	Finally is always associated with try-catch to maintain the cleanup code process.	Finalize is always invoked by garbage collector just before destroying an object to perform clean up code or process.
3)In java final class, we can't inherit. The final method in Java can't be overridden. The final variable in java can't change the value.	The main uses of the Finally block are to clean up the code which is used by the try block.	The finalize method is used to clean up the activities for the object.
4)Final modifier is executed when it is called by the compiler.	Finally, the block is executed after the try-catch block.	Before the object destroy the Finalize method is executed.