Unit 1 (Evolution of JAVA; Variables and Naming Rules)

What is Java?

1: Java is a versatile, object-oriented programming language.
2: It's widely used for developing various types of software, including web applications, mobile applications (Android apps are often written in Java), enterprise systems, and embedded systems.
3: Java's "write once, run anywhere" principle allows programs to run on any device with a Java Virtual Machine (JVM), making it a popular choice for cross-platform development.

Features of JAVA

Java boasts several key features:

1. Platform Independence: Java programs can run on any device with a Java Virtual Machine (JVM), offering platform independence.

2. Object-Oriented: It follows the object-oriented programming (OOP) paradigm, promoting modularity, flexibility, and code reuse.

3. Simple and Familiar: Java syntax is similar to C++, making it easier for programmers to learn, especially those familiar with C-based languages.

4. Automatic Memory Management: Java includes a garbage collector that automatically manages memory, reducing the risk of memory leaks.

5. Multithreading: Java supports concurrent programming with built-in multithreading capabilities, allowing the execution of multiple threads simultaneously.

6. Robust and Secure: Strong type checking, exception handling, and a security model contribute to Java's robustness and security.

7. Dynamic and Extensible: Java supports dynamic loading of classes and dynamic compilation, enhancing flexibility and extensibility.

8. Distributed Computing: It includes features for developing distributed applications, such as Remote Method Invocation (RMI) and Java Naming and Directory Interface (JNDI).

9. Rich Standard Library: Java comes with a comprehensive set of APIs (Application Programming Interfaces) covering various functionalities, making it easy to perform common tasks.

10. High Performance: While not as fast as languages like C or C++, Java's performance has been optimized over the years and is considered satisfactory for most applications.

History/Evolution of Java

1: Java is an Object-Oriented programming language developed by James Gosling at Sun Microsystems in the early 1990s.

2: The team initiated this project called "Green" to develop a language for digital devices such as set-top boxes, television, etc.

3: As the project evolved, it led to the creation of a new programming language, initially called "Oak".

4: In 1995, Sun Microsystems publicly released Oak as Java.

5: Java gained early attention due to its key feature of platform independence.

6: The slogan "Write Once, Run Anywhere" emphasized its ability to run on any device with a Java Virtual Machine (JVM).

7: Sun Microsystems open-sourced Java in 2006 under the GNU General Public License (GPL), making the source code freely available.

8: Oracle Corporation acquired Sun Microsystems in 2010, including ownership of Java.

9: Java 8, released in 2014, introduced significant features like lambda expressions and the Stream API.

10: Java 9 introduced Project Jigsaw, aimed at modularizing the platform. This allowed developers to create more modular and maintainable code.

Difference between C++ and Java

Comparison Factor	Java	C++
Language Type	High-level programming language. Highly efficient for the development of robust and portable applications.	Low-level programming language. Highly efficient for system programming.
Platform Dependency	Platform-independent. With Java, companies can create cross-platform applications efficiently.	Platform-dependent. C++ requires more time to launch cross-platform applications.
Syntax	Java has a simpler and more structured syntax for improved maintenance of complex apps.	C++ has more complex but flexible syntax, allowing to reuse functionality and accelerating complex apps development.
Memory Management	Automatic: increases security of applications.	Manual: offers fine-grained control over memory.
Performance and Speed	Slower app performance, but faster development process and time-to-market.	Higher app performance, but slower development times.
Compatibility	Easy integration with other languages.	Limited compatibility, requires additional tools.
Costs	Lower hiring, development, and maintenance costs.	Lower hardware and performance optimization costs.
Used by Companies	Google, Netflix, Amazon, LinkedIn, eBay, Minecraft, Uber, Spotify, Adobe.	Intel, NASA, BMW, Windows, Chrome, Apple, World of Warcraft, Call of Duty, NVIDIA.

Similarities between C++ and Java

1: Both languages support object-oriented programming.

2: They have the very same type of syntax.

3: The comments syntax is identical between Java and C++.

4: The conditional statements (such as switch, if-else, etc.) and loops (such as for, while etc.) are similar among them both.

5: The relational and arithmetic operators are the same for both of these.

6: The execution of both of these programs begins from the main function.

7: The primitive data types are the same in both.

8: They both have various similar types of keywords.

9: They have multi-threading support.

10: They have pretty similar areas of applications.

JDK

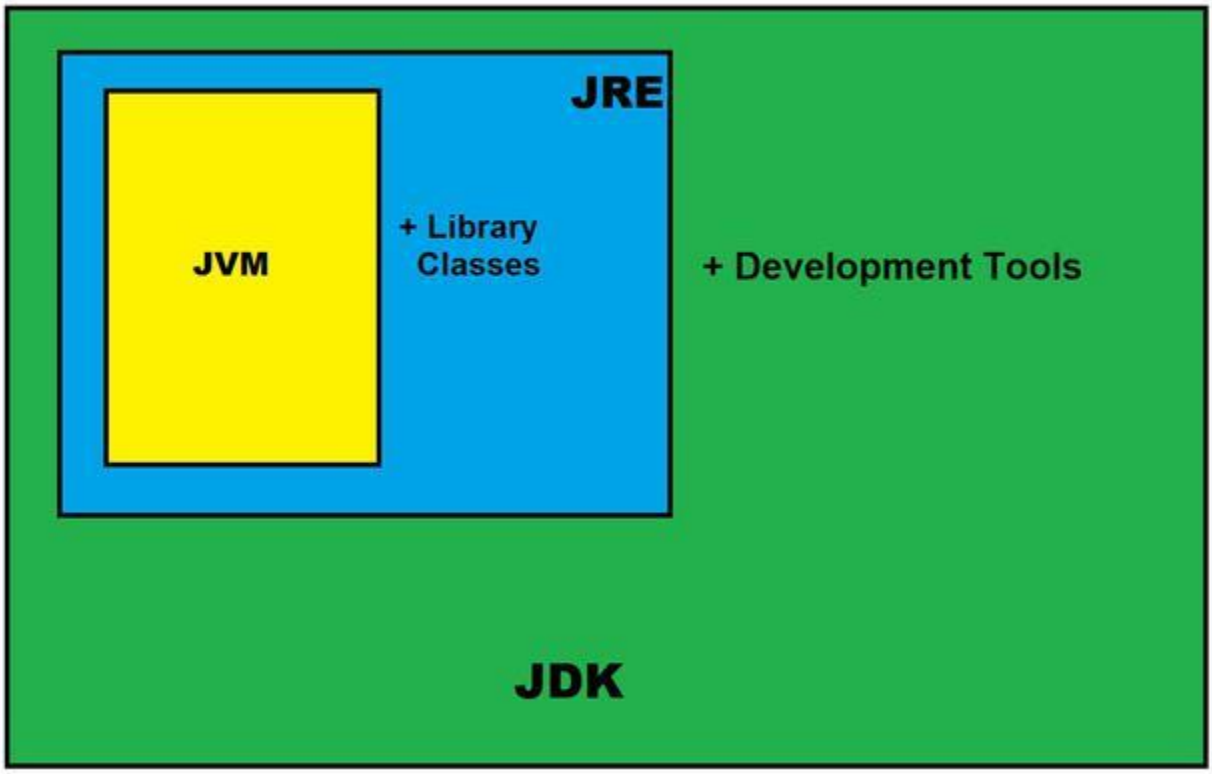
1: JDK stands for Java Development Kit.

2: It is a software development kit used for developing Java applications and applets.

3: The JDK includes the Java Runtime Environment (JRE), necessary for running Java applications, and additional tools such as compilers and debuggers essential for Java programming.

4: Developers use the JDK to write, compile, and run Java code, making it a crucial component in the Java development process.

5: Oracle JDK is the most popular JDK and the main distributor of Java11.



JVM

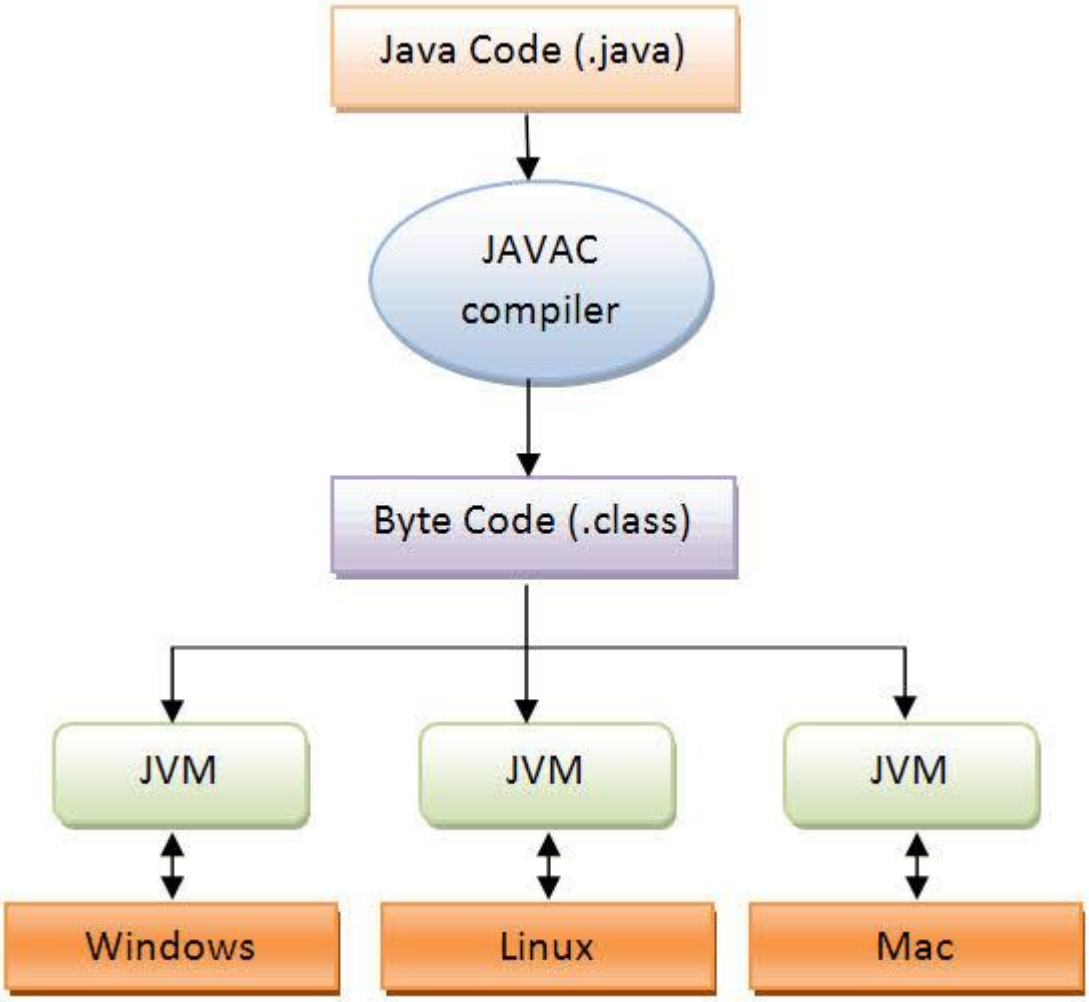
1: A Java virtual machine (JVM) is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode.

2: When you write a Java program, it's translated into a special language called bytecode.

3: The JVM reads and executes this bytecode, making your Java program work on different devices without you having to change the code.

4: It takes care of managing memory and ensures your program runs smoothly and securely.

5: So, the JVM is like a translator that helps your Java code talk to different types of computers.



JRE

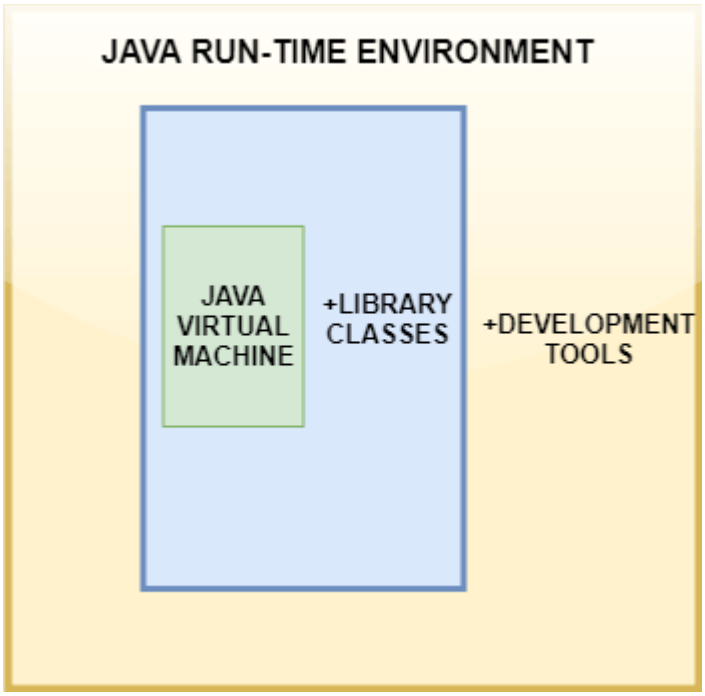
1: JRE stands for Java Runtime Environment.

2: It's a package of software that includes everything needed to run Java applications on your computer.

3: When you want to run a Java program, you need the JRE installed.

4: It contains the Java Virtual Machine (JVM), which interprets and executes Java bytecode, making your Java applications work.

5: The JRE also includes libraries and other components that support the execution of Java programs, but unlike the JDK (Java Development Kit), it doesn't include development tools like compilers.



Class

1: A class is like a blueprint or a template that describes the properties (attributes) and behaviors (methods) that objects created from the class will have.

2: It serves as a model for creating objects.

3: For example, if you were building a car, the class would define what a general car looks like.

4: Here's a simple example of a class in Java:

```
public class Car {
    // Attributes
    String model;
    int year;
```

```
    // Method
    void startEngine() {
        System.out.println("Engine started!");
    }
}
```

Objects

1: An object is an instance of a class.

2: It's a tangible entity that is created based on the structure provided by the class.

3: If a class is a blueprint for a car, an object would be a specific car, like a particular red Ford Mustang.

4: You create objects using the "new" keyword and the constructor of the class.

5: Here's an example:

```
public class Main {
    public static void main(String[] args) {
        // Creating an object of the Car class
        Car myCar = new Car();

        // Setting attributes
        myCar.model = "Mustang";
        myCar.year = 2022;

        // Calling a method
        myCar.startEngine();
    }
}
```

Instantiation in Java

Instantiation in Java refers to the process of creating an instance (object) of a class. It involves allocating memory for the object and initializing its attributes. Here's how you instantiate an object in Java:

Syntax:

```
ClassName objectName = new ClassName();
```

Replace ClassName with the name of the class, and objectName with the name you give to the instance of that class

Example:

```
// Assume we have a simple class called Dog
public class Dog {
    String breed;
    int age;

    void bark() {
        System.out.println("Woof!");
    }
}

// Now, let's instantiate an object of the Dog class
public class Main {
    public static void main(String[] args) {
        // Instantiating a Dog object
        Dog myDog = new Dog();

        // Setting attributes
        myDog.breed = "Labrador";
        myDog.age = 3;

        // Calling a method
        myDog.bark();
    }
```

In this example, myDog is an instance of the Dog class created through the new keyword and the Dog() constructor.

Variables in java

1: In Java, variables are containers for storing data values. They have a specific data type, which defines the type of data they can hold.

2: Before using a variable, you need to declare it. This involves specifying the variable's name and its data type.

3: After declaring a variable, you can assign an initial value to it. This is called initialization.

4: Alternatively, you can combine declaration and initialization in a single line:

```
int age = 25; // Declaration and initialization in one line
```

5: Variable names must follow certain rules, such as starting with a letter, being case-sensitive, and avoiding reserved keywords.
6: The scope of a variable determines where in the code it can be accessed. Variables can be local (within a method or block), instance (belonging to an object), or class (shared among all instances of a class).
7: You can use the `final` keyword to declare a variable as a constant, meaning its value cannot be changed after initialization.

Java data types

Java has two categories of data types: primitive and reference.

Primitive data types include:

1. byte: 8-bit integer

2. short: 16-bit integer

3. int: 32-bit integer

4. long: 64-bit integer

5. float: 32-bit floating-point

6. double: 64-bit floating-point

7. char: 16-bit Unicode character

8. boolean: Represents true or false values

Reference data types include:

1. Objects: Instances of classes

2. Arrays: Collections of elements of the same type

Operators

Java supports various types of operators, categorized into:

1. Arithmetic Operators:

- `+` (addition)
- `-` (subtraction)
- `*` (multiplication)
- `/` (division)
- `%` (modulo)

2. Comparison Operators:

- `==` (equal to)
- `!=` (not equal to)
- `<` (less than)
- `>` (greater than)
- `<=` (less than or equal to)
- `>=` (greater than or equal to)

3. Logical Operators:

- `&&` (logical AND)
- `||` (logical OR)
- `!` (logical NOT)

4. Assignment Operators:

- `=` (assign)
- `+=` (add and assign)
- `-=` (subtract and assign)
- `*=` (multiply and assign)
- `/=` (divide and assign)
- `%=` (modulo and assign)

5. Increment/Decrement Operators:

- `++` (increment by 1)
- `--` (decrement by 1)

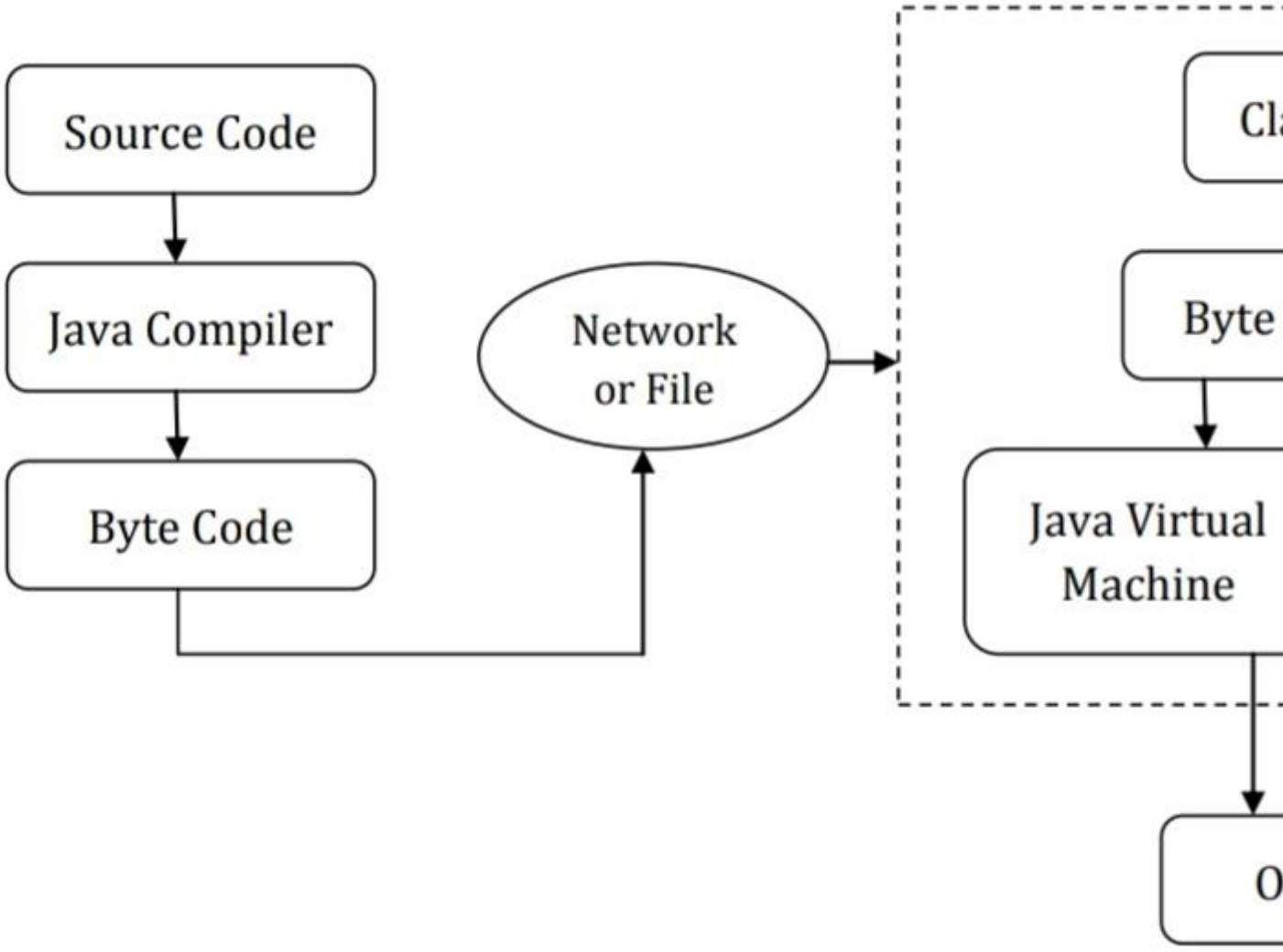
6. Bitwise Operators:

- `&` (bitwise AND)
- `|` (bitwise OR)
- `^` (bitwise XOR)
- `~` (bitwise NOT)
- `<<` (left shift)

- `>>` (right shift)
- `>>>` (unsigned right shift)

These operators play a crucial role in performing various operations in Java programs.

Java Environment Architecture



Bytecode: As discussed above, javac compiler of JDK compiles the java source code into bytecode so that it can be executed by JVM. The compiler saves the the bytecode in a .class file.

Class Loader: The Java Classloader is a part of the Java Run-time Environment

that is responsible for dynamically loading of Java classes into the Java Virtual Machine.

Byte Code Verifier: Is again a part of Java Run-time Environment, after loading

the bytecode in JVM, bytecode are first inspected by a verifier. The byte code verifier also checks that the obviously damaging instructions cannot perform actions.

Java Virtual Machine (JVM): Java Compiler takes the java program (.java file) as input and generates java bytecode also called as class file (.class) as output. At last, JVM executes the bytecode (.class file) generated by the compiler. This is called the program run phase.

The Just-In-Time (JIT) compiler: translates bytecode into native machine code at runtime. JIT compilation enables Java to maintain platform independence while achieving efficient execution on specific hardware.

Types of variables

In Java, variables can be classified into three main types:

1. Local Variables:

- Declared within a method, constructor, or block.
- Limited to the scope in which they are declared.
- Must be initialized before use.

void exampleMethod() {

```
int localVar = 10; // Local variable
        // ...
    }
```

2. Instance Variables (Non-Static Fields):

- Belong to an instance of a class (object).
  - Each object has its own copy of these variables.
  - They are initialized when the object is created.

class MyClass {

```
int instanceVar; // Instance variable
    }
```

3. Class Variables (Static Fields):

- Associated with the class rather than instances.
-Shared among all instances of the class.
- Initialized when the class is loaded.

class AnotherClass {

```
static int classVar; // Class variable
```

Scope of Variables

Scope of a variable is the area or region in the program where the variable is accessible. in Java, Scope of a variable can determined at compile time and independent of function call stack.

1. Class Scope (Class Variable):

-Declaration: Declared within a class, marked with `static` keyword.

- Scope: Exists for the entire duration of the program.

- Access: Shared among all instances of the class.

```
public class Example {

      static int classVar; // Class variable
  }
```

2. Method Scope (Local Variable):

- Declaration: Declared within a method.
- Scope: Limited to the method where it is declared.
- Access: Not accessible outside the method.

```
public class Example {
      public void someMethod() {
          int localVar = 10; // Local variable
      }
  }
```

3. Loop Scope (Local Variable in a Loop):
- Declaration: Inside the initialization of a loop.
- Scope: Limited to the loop body.
- Access: Not accessible outside the loop.

```
public class Example {
      public void loopExample() {
          for (int i = 0; i < 5; i++) {
              // i is a local variable with loop scope
          }
      }
  }
```

4. Bracket (Block) Scope (Local Variable in a Block):
- Declaration: Inside a block of code enclosed by curly braces `{}`.
- Scope: Limited to the block where it is declared.
- Access: Not accessible outside the block.

```
public class Example {
      public void blockExample() {
          {
              int blockVar = 20; // Local variable with block scope
```

```
            // blockVar is only accessible within this block
        }
    }
}
```

Garbage collection

1: Java garbage collection is the process to perform automatic memory management which is done by java programs.

2: While executiong Java programs on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program.

3: Eventually, some objects will no longer be needed.

4: The garbage collector identifies such unused objects so that memory can be freed by deleteing them.

5: Garbage Collection is the process to free the memory occupied by unused objects.

6: The gc() method is used to call the garbage collector to perform cleanup processing. The gc() method is found in System and Runtime classes.

7: The finalize() method in Java allows an object to clean up resources before it is garbage collected.

Source File Declaration Rules In Java

1. There can be only one public class per source code file.

2. Comments can appear at the beginning or end of any line in the source code file.

3. If there is a public class in a file, the name of the file must match the name of the public class.

4. If the class is part of a package, the package statement must be the first line in the source code file, before any import statements that may be present.

5. If there are import statements, they must go between the package statement.

If there isn’t a package statement, then the import statement(s) must be the first line(s) in the source code file.

6. import and package statements apply to all classes within a source code file.

7. A file can have more than one nonpublic class.

8. Files with no public classes can have a name that does not match any of the classes in the file.

---

naming conventions

1. Classes Naming Conventions:

- Use nouns or noun phrases.
- Start with an uppercase letter.
- Use camel case (e.g., `MyClass`, `PersonDetails`).

2. Interfaces Naming Conventions:

- Use adjectives or adjective phrases.
- Start with an uppercase letter.
- Use camel case (e.g., `Runnable`, `Serializable`).

3. Methods Naming Conventions:

- Use verbs or verb phrases.
- Start with a lowercase letter.

- Use camel case (e.g., `calculateTotal`, `getUserInfo`).

4. Variables Naming Conventions:

- Use meaningful names that reflect the purpose.
- Start with a lowercase letter.
- Use camel case (e.g., `itemName`, `totalAmount`).

5. Constants Naming Conventions:

- Use uppercase letters with underscores separating words.
- Often static and final.
- Example: `MAX_SIZE`, `PI_VALUE`.

6. Enumeration Naming Conventions:

- Use uppercase letters.
- Enum constants are typically all

uppercase.

- Use underscores to separate words.
- Example: `DayOfWeek.MONDAY`, `*Color.RED/*`.

7. Packages naming conventions:
-Use all lowercase for package names.
-Start with the reverse of your domain name.

Adhering to these conventions improves code readability and consistency, making it easier for developers to understand and maintain Java code.

Unit 2 (Decision Making and Looping)

if Statement

The Java if statement tests the condition. It executes the if block if the mentioned condition is true else it will exit the if block.

Syntax:

```
if(condition){
//code/ to be executed }
```

Example:

```
//Java/ Program to demonstrate the use of if statement.
public class IfExample {
public static void main(String[] args) {
 //defining/ an 'age' variable
 int age=20;
 //checking/ the age
 if(age>18){
 System.out.print("Age is greater than 18");
 }
}
```

ck.

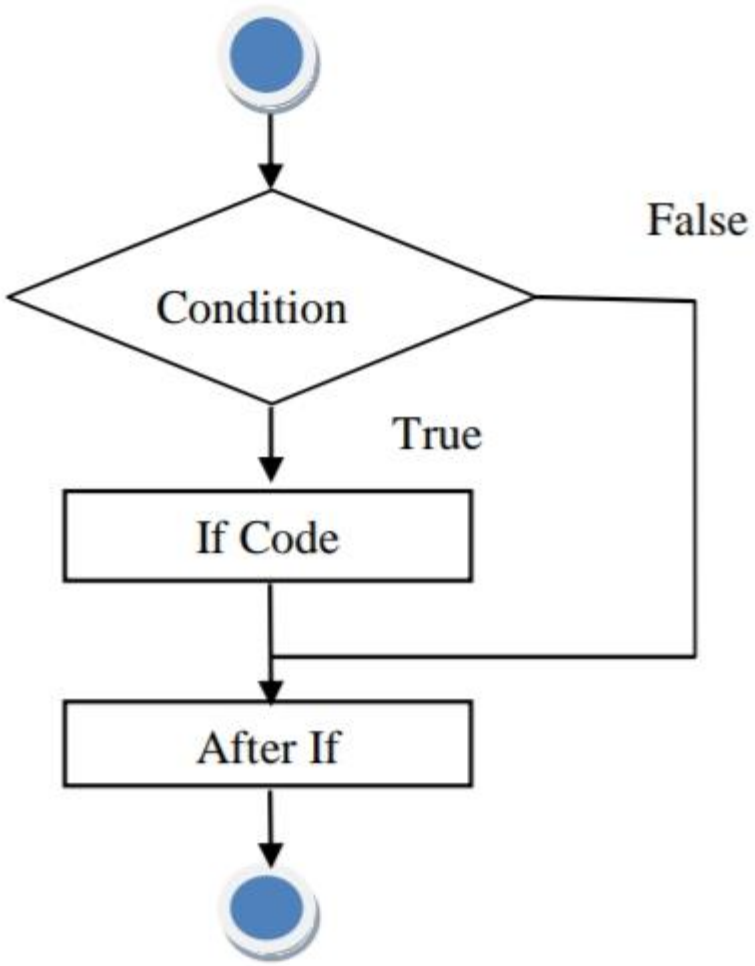


Fig. 2.1 : Java if statement Execution steps

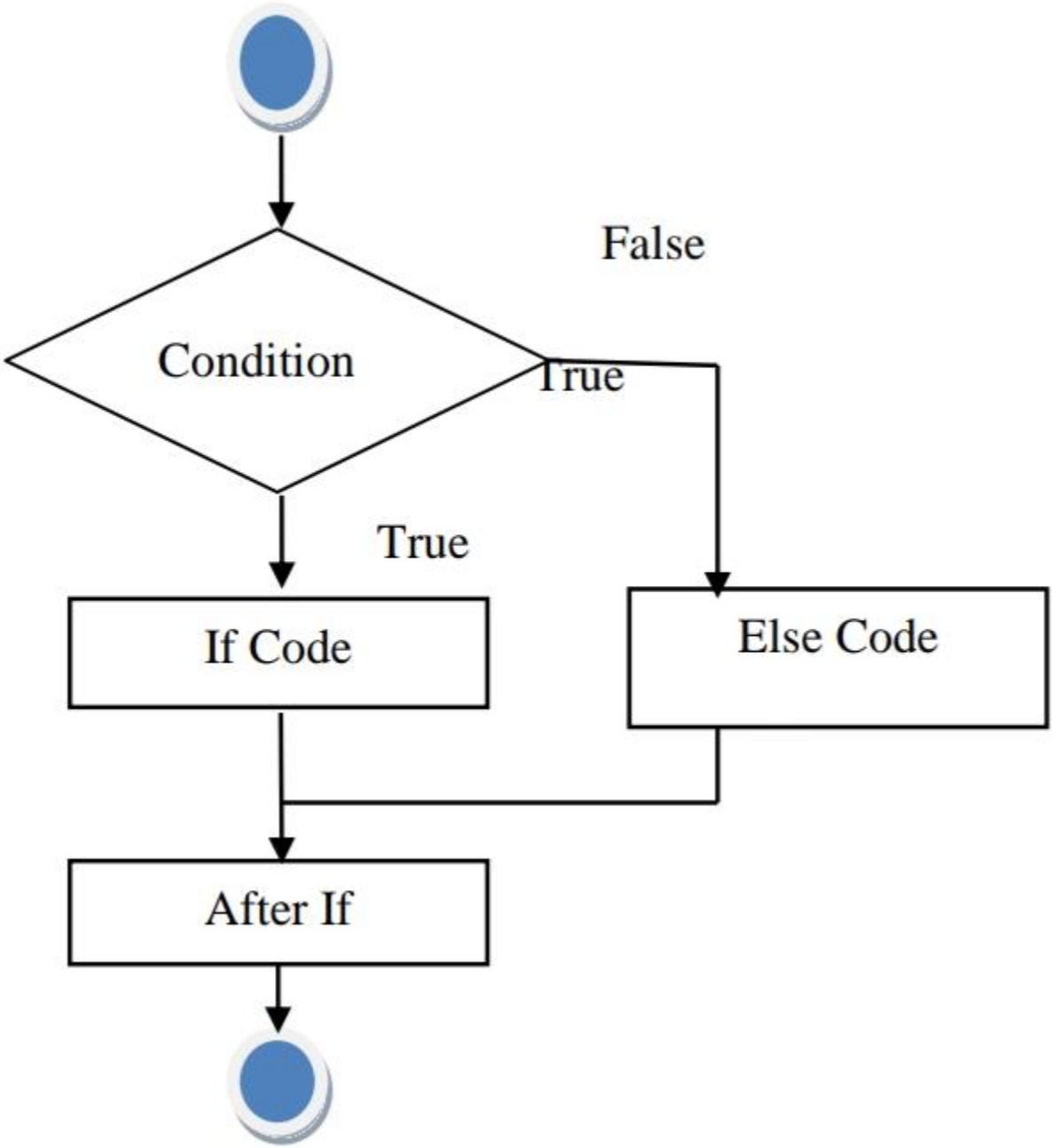
---

if-else Statement

The Java if-else statement also tests the condition. It executes the if block if condition is true otherwise else block is executed.

Syntax:

```
if(condition){
//code/ if condition is true
}else{
//code/ if condition is false
}
```



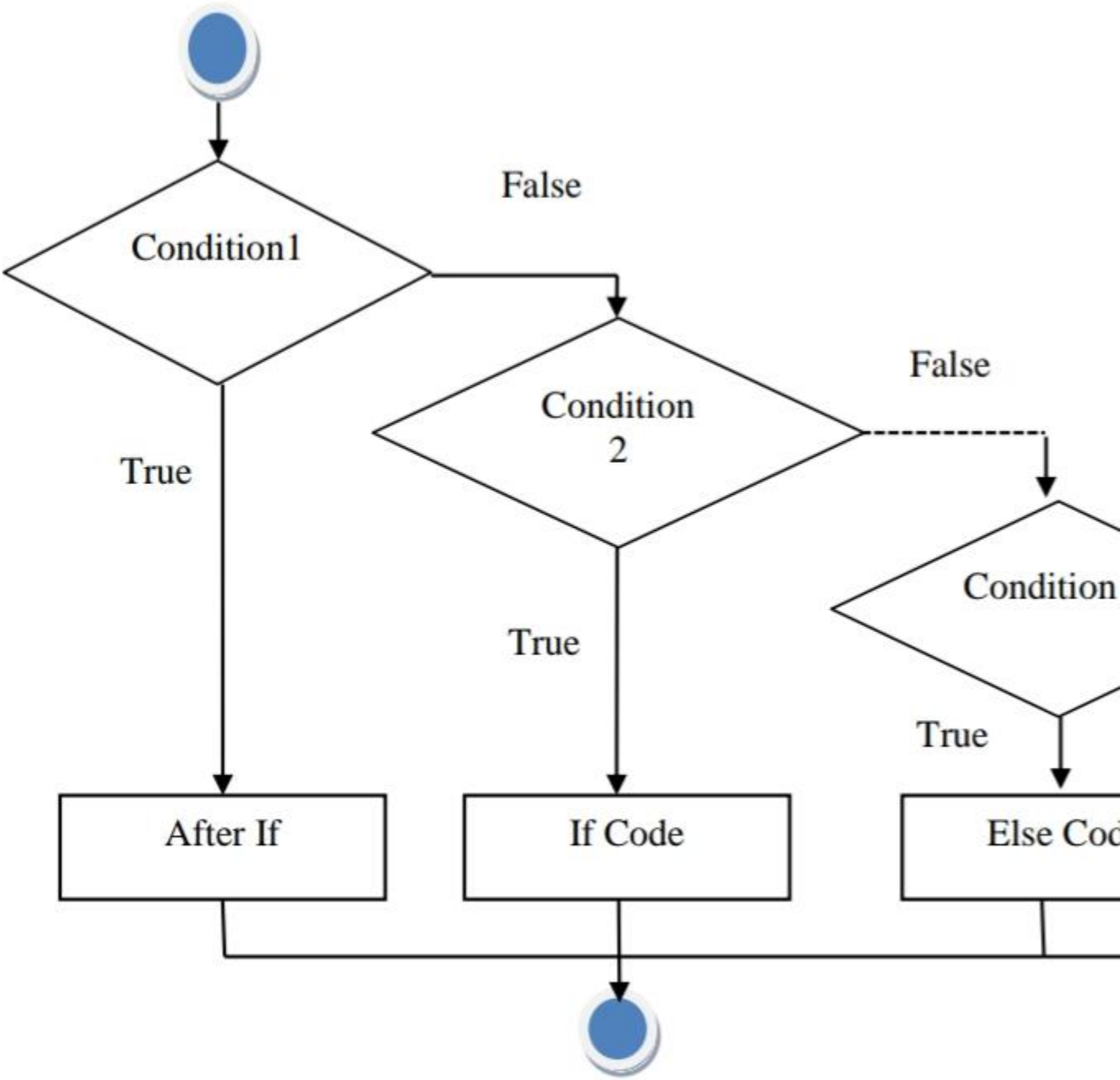
if-else-if ladder

The if-else-if hierarchy statement executes only one condition from multiple statements. It can be used when you have to check multiple conditions, and based on satisfied condition need to check

associated statements. It will execute the single block of statements associated with condition which get satisfied.

Syntax:

```
if(condition1){
//This/ code will be executed if condition 1is true
}else if(condition2){
//This/ code will be executed if condition2 is true
}
else if(condition3){
//This/ code will be executed if condition3 is true
}
...
else{
//This/ code will be executed if all the conditions are false
}
```


Condition1
False
True
Condition
2
False
True
Condition
True
After If
If Code
Else Cod

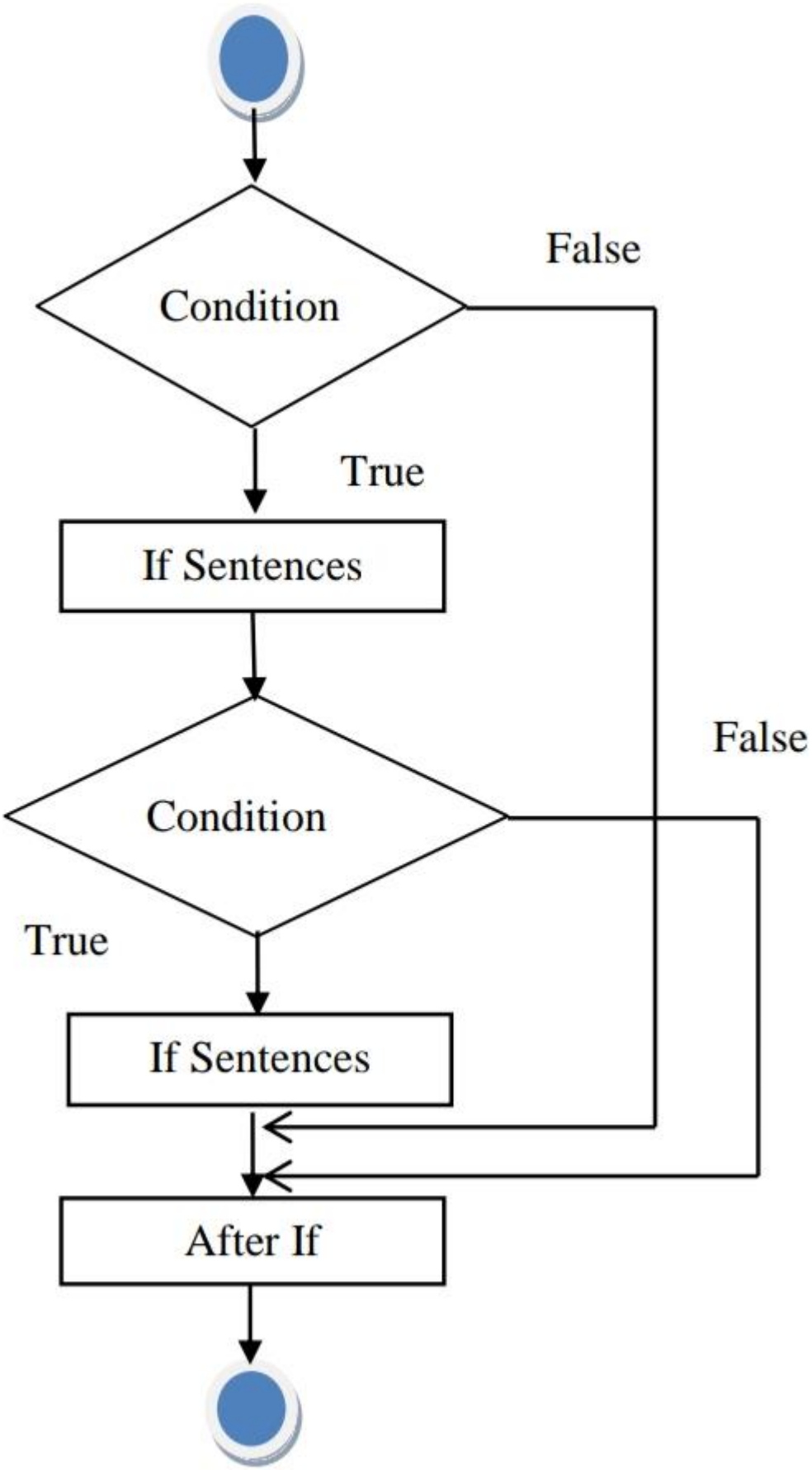
Nested if

The nested if statement represents the if block within another if block. Here, the inner if block

condition executes only when outer if block condition is true.

Syntax:

```
if(condition){
 //code/ to be executed
 if(condition){
 //code/ to be executed
 }
}
```


Condition
False
True
If Sentences
Condition
False
True
If Sentences
After If

Ternary Operator

We can also use ternary operator (? :) to perform the task of if...else statement. It is a simplest

way to check the condition. If the condition is true then resultsof'?'is returnedBut, if the condition

is false, the result of ':' is returned.

Example:

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Try programiz.pro");

        int num1=10;
        int num2=20;

       int result=(num1>num2)? (num1+num2):(num1-num2);

       System.out.println(result);
    }
}
```

For Loop

When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop:

Syntax:

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
```

```
}
```
Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

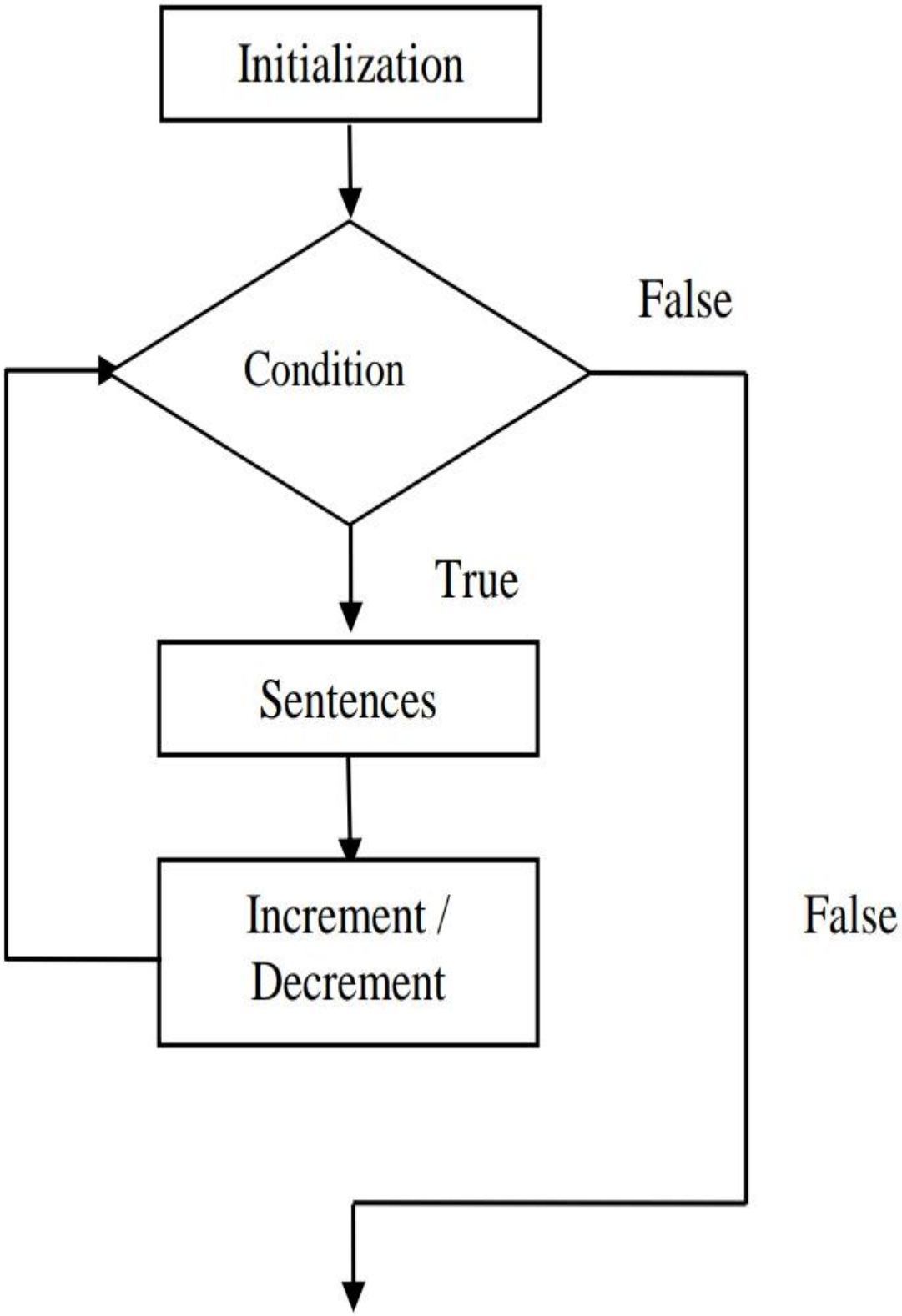


Fig. 2.5 : Java for loop Execution steps

Nested For Loop

Writing for loop inside the loop, it is known as nested for

loop. The inner loop executes completely whenever outer loop executes.

Example:

```
// Outer loop
for (int i = 1; i <= 2; i++) {
  System.out.println("Outer: " + i); // Executes 2 times

  // Inner loop
  for (int j = 1; j <= 3; j++) {
    System.out.println(" Inner: " + j); // Executes 6 times (2 * 3)
  }
}
```

While Loop

A while loop is a control flow statement that repeats code based on a Boolean condition.

Syntax:

```
while(condition){
//code/ to be executed
}
```

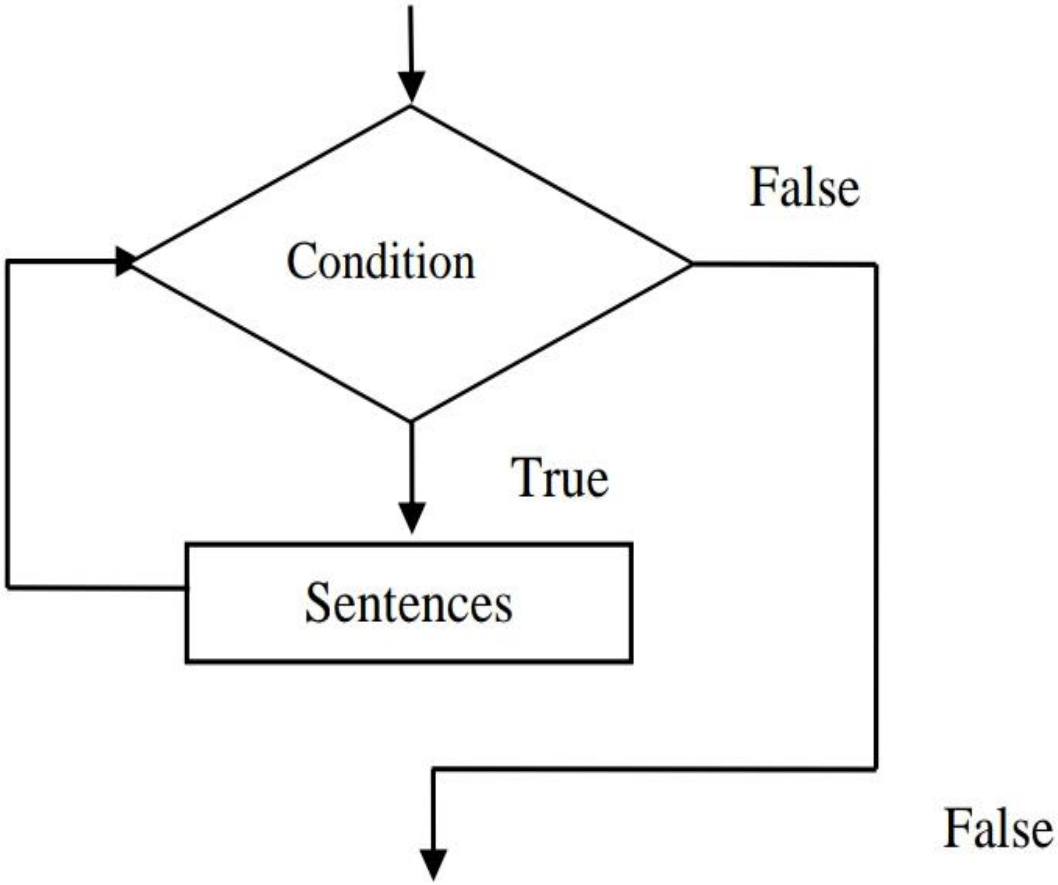


Fig. 2.6 : Java while loop Execution steps

---

do-while Loop

A do while loop is a control flow statement that runs while a logical expression is true. It executes a block of code and then either repeats the block or exits the loop depending on a given boolean condition.

Syntax:

```
Do{
```

```
//code/ to be executed
}while(condition);
```

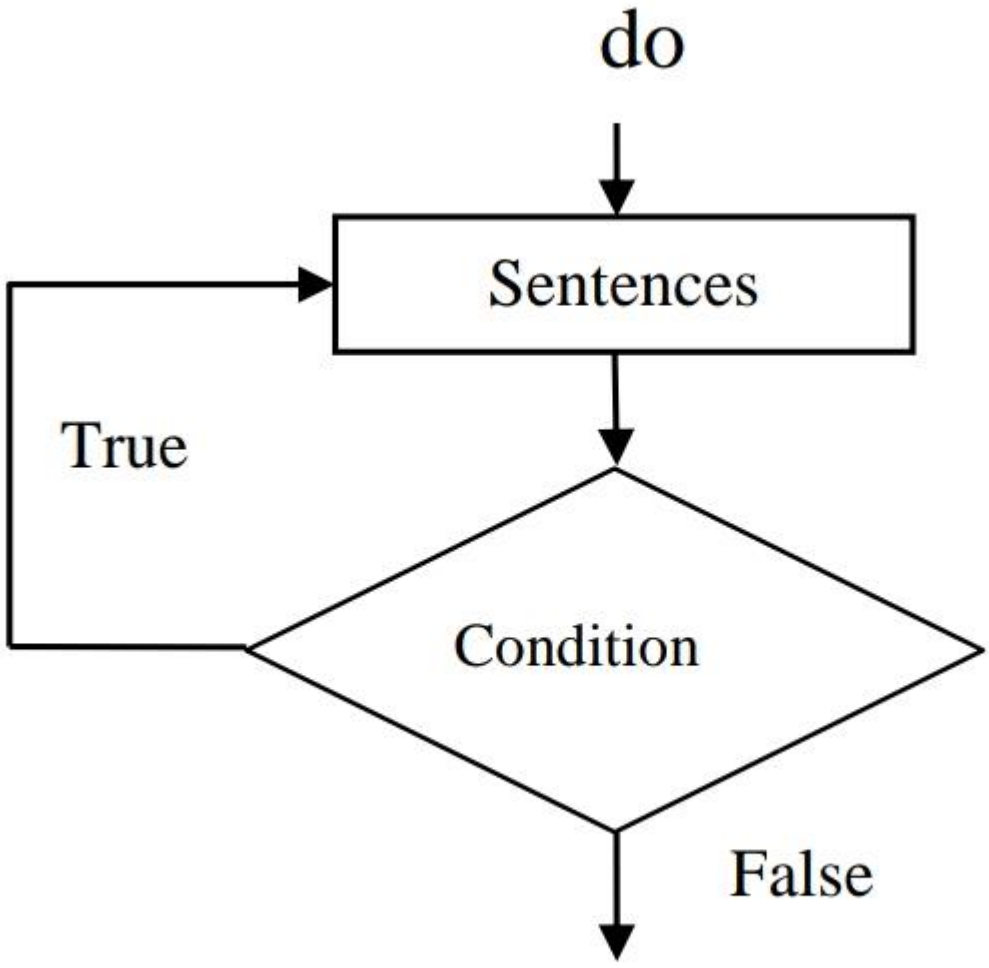


Fig. 2.7 : Java do while loop Execution steps

Switch Statement

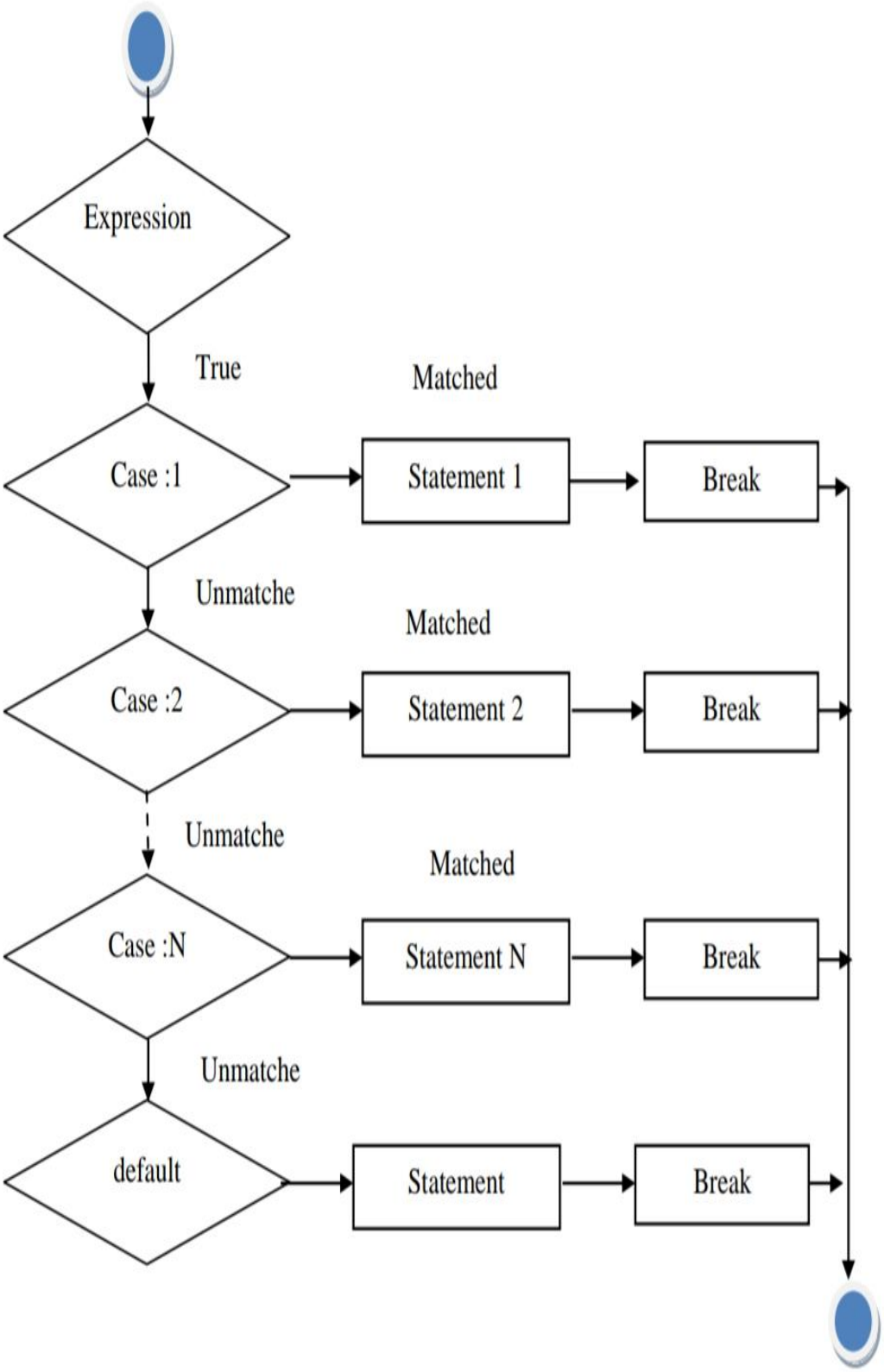
Instead of writing many if..else statements, you can use the switch statement. The switch statement selects one of many code blocks to be executed:

This is how it works:

1. The switch expression is evaluated once.
2. The value of the expression is compared with the values of each case.
3. If there is a match, the associated block of code is executed.
4. The break and default keywords are optional.

Syntax :

```
switch(expression){ case value1: //code/ to be executed; break;
//optional/ case value2: //code/ to be executed; break; //optional/
......  default: code to be executed if all cases are not matched; }
```


Expression
True
Matched
Case :1
Statement 1
Break
Unmatche
Matched
Case :2
Statement 2
Break
Unmatche
Matched
Case :N
Statement N
Break
Unmatche
default
Statement
Break

Break Statement

1. A break statement is used to terminate the currently executing loop.
2. Once the break statement encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
3. The Java break statement is used to break currently executing loop or switch statement.
4. It breaks the current flow of the program on satisfying the specified condition.
5. In case if it is written in inner loop, it breaks only the inner loop.
6. We can use Java break statement in all types of loops such as for loop, while loop and do-while loop.

Example:

```
//Java/ Program to demonstrate the use of break statement inside the
for loop.
public class BreakExample {
public static void main(String[] args) {
 //using/ for loop
 for(int i=1;i<=10;i++){
 if(i==5){
 //breaking/ the loop
 break;
 }
 System.out.println(i);
 } } }
```

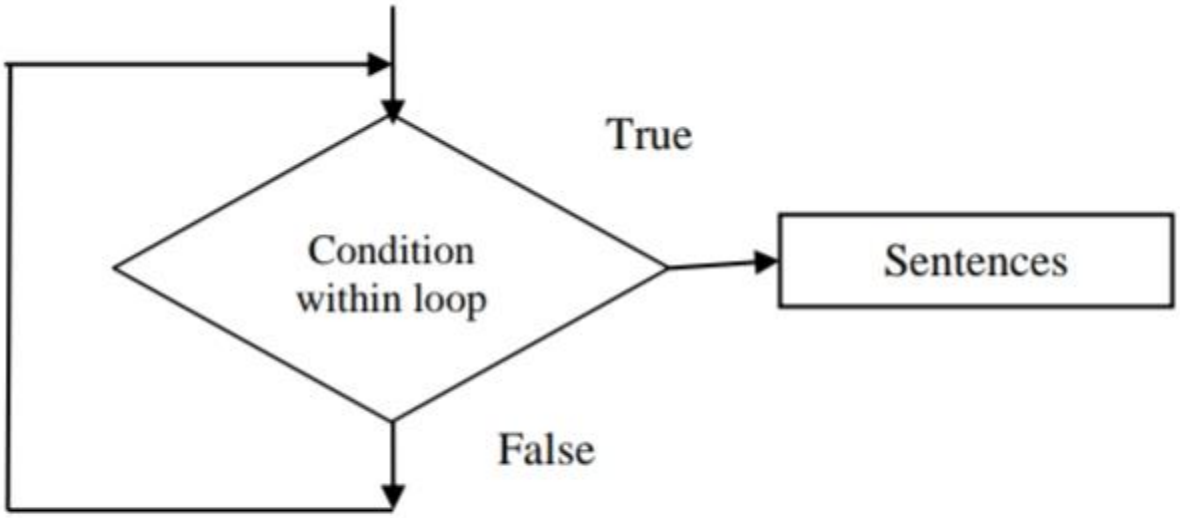


Fig. 2.9: Java break statement Execution steps

Continue Statement

1. The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately.
2. It can be used with for loop or while loop.
3. The Java continues statement is used to continue the loop.
4. It continues the current flow of the program and skips the remaining code at the specified condition.
5. In case of an inner loop, it continues the inner loop only.

Example:

```
//Java/ Program to demonstrate the use of continue statement //inside/
the for loop. public class ContinueExample { public static void
main(String[] args) {  //for/ loop  for(int i=1;i<=10;i++){  if(i==5){
//using/ continue statement  continue;//it/ will skip the rest
statement  }  System.out.println(i);  } }}
```

Types of Java Comments

1. Java Single Line Comment: The single line comment is used to comment only one line.

// This is a single line comment

2. Java Multi Line Comment: The multi line comment is used to comment multiple lines of code.

/*

This is

multi line

comment

*/

3. Java Documentation Comment: The documentation comment are the type of comment which are used to create documentation API. To create documentation API we require

to use javadoc tool.

/*This is documentation comment */

Unit 3 (Implementation of Methods)

Method Signature

In Java, a method signature consists of the method's name, parameter types, and their order. It excludes the method's return type. The return type of method and exceptions are not considered as part of method signature.

For example:

```
void methodName(int parameter1, String parameter2)
```

Constructor

1. In Java, a constructor is a block consisting of executable sentences or instructions just similar to the method.

2. It is called automatically when an instance of the class (object) is created, and memory is allocated for the object.

3. It is named as constructor because it constructs the values at the time of object creation.

4. Writing a constructor for a class is not nessesary, class may have zero

or any number of constructors.

5. If your class doesn‟t have any constructor java compiler creates a default constructor for your class.

6. You can say it is a special type of method which is used to initialize the object.

7. When an object is created using new () keyword the constructor gets called.

8. It calls a default constructor.

9. After execution of object creation statement constructor gets called

and starts its execution.

No-argument constructor

1. A constructor that has no parameter is known as default

constructor.

2. If we don‟t define a constructor in a class, then compiler creates default constructor (with no arguments) for the class.

3. And if we write a constructor with arguments or no-arguments

then the compiler does not create a default constructor.

4. Depending on the type Default constructor provides the default values to the object like 0, null, etc.

5. A constructor which doesn‟t have ant parameters is called as “Default Constructor”.

Example:

```
class Employee{
Employee() //creating/ a default constructor
{
System.out.println(“Employee Class Default Constructor”);}
public static void main(String args[]){ //main/ method
Employee b=new Employee (); //calling/ a default constructor
} }
```

Parameterized Constructor

1. A constructor which has a finite number of parameters is called a parameterized constructor.

2. It is used if we want to initialize fields of the class with your own values, and then use a parameterized constructor.

3. The parameterized constructor provides different values to the distinct objects.

4. However, you can provide the same values also.

Example:

```
class ConstrucrorDemo
{
int a;
String s;
ConstrucrorDemo()
{
System.out.println("U r in Default Constructor");
a=10;
s="Te comp";
}
ConstrucrorDemo(int a,String s)
{
System.out.println("U r in Parametric Constructor");
this.a=a;
this.s=s;
}
void display()
{
System.out.println("U r in function now");
System.out.println("The value of A is==>"+a);
System.out.println("The value of S is==>"+s);
}
public static void main(String[] args)
{
System.out.println("Hello World!");
ConstrucrorDemo c=new ConstrucrorDemo();
System.out.println("Using the ref of Default Constructor Values
are==>");
c.display();
ConstrucrorDemo c1=new ConstrucrorDemo(20,"T3 Batch");
System.out.println("Using the ref of Parametric Constructor Values
are==>");
c1.display();
}
}
```

Copy constructor

In Java, a copy constructor is a constructor that creates a new object by copying the values of an existing object. It typically takes an object of the same class as a parameter and initializes the new object's state with the values from the existing object. Here's an example

```
public class MyClass {
    private int value;

    // Copy constructor
    public MyClass(MyClass other) {
        this.value = other.value;
    }

    // Constructor
    public MyClass(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }

    public static void main(String[] args) {
        MyClass obj1 = new MyClass(10);
        MyClass obj2 = new MyClass(obj1); // Using the copy
constructor
        System.out.println("Value of obj1: " + obj1.getValue());
        System.out.println("Value of obj2: " + obj2.getValue());
    }
}
```

In this example, the MyClass copy constructor public MyClass(MyClass other) takes an object of type MyClass as a parameter and initializes the new object with the same value.

Method Overloading

1. Method overloading in Java allows you to define multiple methods in the same class with the same name but with different parameters.

2. This enables you to perform similar operations with different inputs or to provide more flexibility to users of your class.

Here's an example

```
public class Calculator {
    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Method to add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method to add two doubles
    public double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator calculator = new Calculator();

        // Calling the add method with different parameters
        System.out.println("Sum of 5 and 3 is: " + calculator.add(5,
3));
        System.out.println("Sum of 5, 3, and 7 is: " +
calculator.add(5, 3, 7));
        System.out.println("Sum of 5.5 and 3.7 is: " +
calculator.add(5.5, 3.7));
    }
}
```

In this example, the Calculator class defines three add methods, each with a different number or type of parameters. When you call the add method, Java determines which version of the method to execute based on the number and types of arguments passed to it.

Constructor overloading

1. In Java, overloaded constructor is called based on the parameters specified when a new is executed.

2. Sometimes there is a need of initializing an object in different ways. This can be done using constructor overloading.

3. this() reference can be used during constructor overloading to call the default constructor implicitly from the parameterized constructor.

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

    // Constructor with name parameter
    public Person(String name) {
        this.name/ = name;
    }

    // Constructor with name and age parameters
    public Person(String name, int age) {
        this.name/ = name;
        this.age = age;
    }

    // Getter methods
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public static void main(String[] args) {
        // Creating objects using different constructors
        Person person1 = new Person("Alice");
        Person person2 = new Person("Bob", 30);

        // Displaying information
        System.out.println("Person 1: Name - " + person1.getName() +
", Age - " + person1.getAge());
```

```
        System.out.println("Person 2: Name - " + person2.getName() +
", Age - " + person2.getAge());
    }
}
```

In this example, the Person class has two constructors: one that takes only a name parameter and another that takes both a name and an age parameter. Depending on how you create a Person object, Java will invoke the appropriate constructor.

Method Overriding

1. If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.

2. In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

3. Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.

4. Method overriding is used for runtime polymorphism.

5.The method must have the same name as in the parent class.

6. The method must have the same parameter as in the parent class.

7. There must be an IS-A relationship (inheritance).

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

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
```

```
    public static void main(String[] args) {
        Animal animal = new Animal();
        animal.makeSound(); // Output: Animal makes a sound

        Dog dog = new Dog();
        dog.makeSound(); // Output: Dog barks

        // Polymorphic behavior
        Animal anotherDog = new Dog();
        anotherDog.makeSound(); // Output: Dog barks
    }
}
```

In this example, the Dog class overrides the makeSound() method from the Animal class. When you call makeSound() on a Dog object, it executes the Dog class's implementation of the method. The @Override annotation is used to indicate that the makeSound() method in the Dog class is overriding the makeSound() method in the Animal class.

Final keyword

In Java, the final keyword can be applied to variables, methods, and classes, and its meaning varies depending on where it is used:

1. Final Variables:

When applied to a variable, it means the variable's value cannot be changed once assigned.

For primitive data types, it means the value cannot be changed. For reference types, it means the reference cannot be changed (though the object itself can still be modified if it is mutable).

2. Final Methods:

When applied to a method, it means the method cannot be overridden by subclasses.

3. Final Classes:

When applied to a class, it means the class cannot be subclassed (i.e., it cannot have any subclasses).

```
class Example {
    final int constant = 10;
    final String message = "Hello";

    final void finalMethod() {
        // Method implementation
    }
}

final class FinalClass {
    // Class implementation
}

// This will result in a compile-time error because you cannot
subclass a final class
// class SubClass extends FinalClass {}
```

---

static keyword

1. The static keyword in Java is used for memory management mainly.

2. We can apply static keyword with variables, methods, blocks and nested classes.

3. The static keyword belongs to the class than an instance of the class.

The static can be:

Variable (also known as a class variable)

Method (also known as a class method)

Block

Nested class

4. Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created.

5. All students have its unique rollno and name, so instance data member is good in such case.

6. Here, "college" refers to the common property of all objects.

7. If we make it static, this field will get the memory only once.

---

this keyword

1. In Java, ‘this’ is a reference variable that refers to the current object, or can be said “this” in Java is a keyword that refers to the current object instance.

2. It can be used to call current class methods and fields, to pass an instance of the current class as a parameter, and to differentiate between the local and instance variables.

3. Using “this” reference can improve code readability and reduce naming conflicts.

4.In/ Java, this is a reference variable that refers to the current object on which the method or constructor is being invoked.
5. It can be used to access instance variables and methods of the current object.

```
public class Person {


    // Fields Declared
```

```
    String name;
    int age;

    // Constructor
    Person(String name, int age)
    {
        this.name/ = name;
        this.age = age;
    }

    // Getter for name
    public String get_name() { return name; }

    // Setter for name
    public void change_name(String name)
    {
        this.name/ = name;
    }

    // Method to Print the Details of
    // the person
    public void printDetails()
    {
        System.out.println("Name: " + this.name/);
        System.out.println("Age: " + this.age);
```

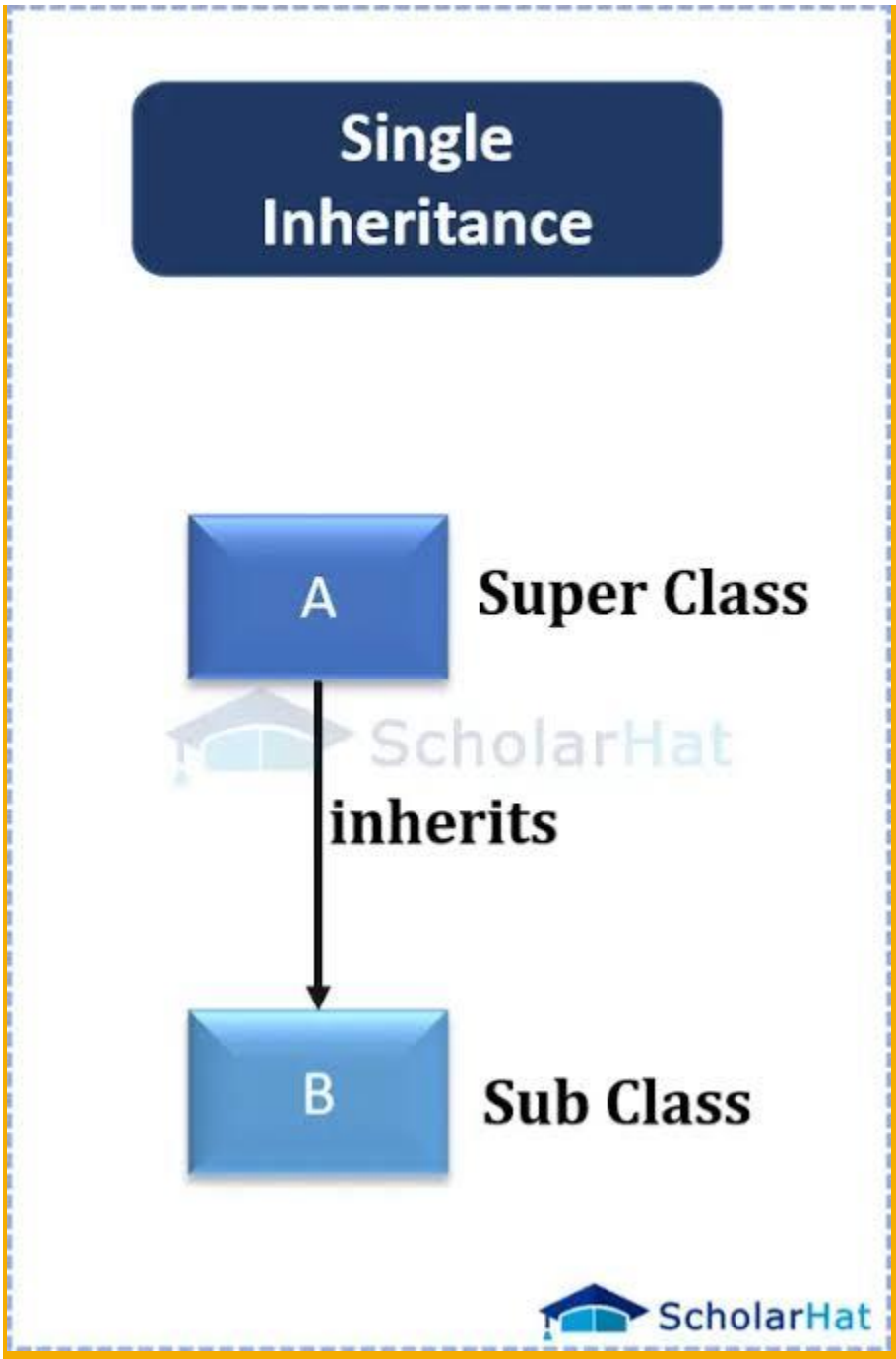
```
        System.out.println();
    }

    // main function
    public static void main(String[] args)
    {
        // Objects Declared
        Person first = new Person("ABC", 18);
        Person second = new Person("XYZ", 22);

        first.printDetails();
        second.printDetails();

        first.change_name("PQR");
        System.out.println("Name has been changed to: "
                           + first.get_name());
    }
```

Single Inheritance

Single inheritance in Java refers to the capability of a class to inherit properties and behaviors from only one superclass. In Java, each class can extend only one other class. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

```
class Animal{
void eat(){
```

```
System.out.println("eating...");
}
}
class Dog extends Animal{
void bark()
{
System.out.println("barking...");
}
}
class TestInheritance
{
public static void main(String args[])
{
Dog d=new Dog();
d.bark();
d.eat/();
}
}
```



Multiple inheritance

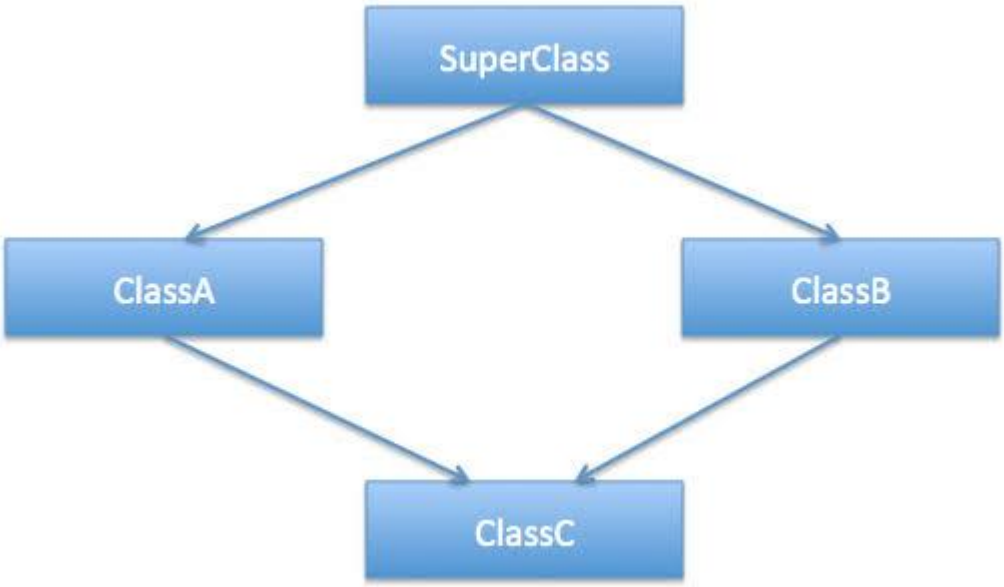
1. Multiple inheritance in Java refers to a scenario where a class can inherit properties and methods from more than one superclass.

2. However, Java does not support multiple inheritance directly through classes due to the complexity and ambiguity it can cause.

3. One example of this complexity is the 'Diamond Problem,' where a class inherits from two classes that have a common ancestor, leading to ambiguity regarding which inherited method should be called.

4.However, you can achieve similar functionality using interfaces. Here's an example:

```
interface A {
    void methodA();
}

interface B {
    void methodB();
}

class MyClass implements A, B {
    public void methodA() {
        System.out.println("Method A");
    }

    public void methodB() {
        System.out.println("Method B");
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.methodA();
        obj.methodB();
    }
}
```



Hierarchical Inheritance

1. One superclass and a number of subclasses are included in Java's hierarchy of inheritance.

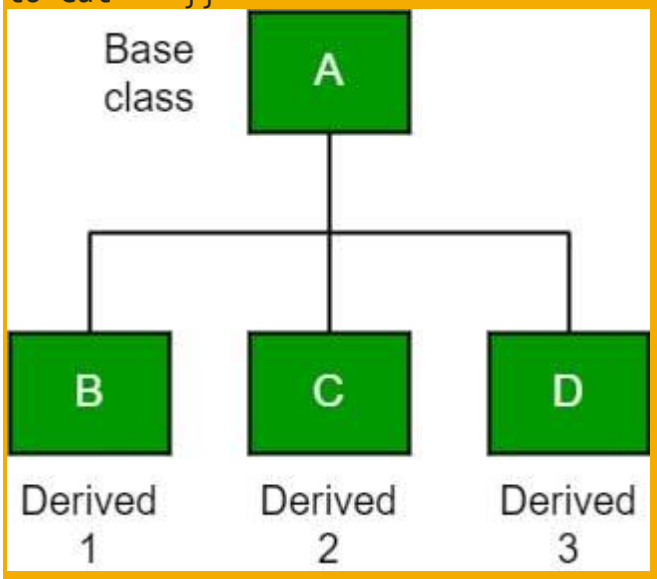
2. There must be at least two subclasses for inheritance to take place.

3. When a superclass and numerous subclasses that will inherit from it exist, inheritance can function.

4. To allow for the flow of properties during inheritance, a superclass must be constructed first, followed by various subclasses.

```
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");    }}public
class Main {    public static void main(String[] args) {        Dog
dog = new Dog();        dog.eat/();  // Inherited from Animal
dog.bark(); // Specific to Dog                Cat cat = new Cat();
```

```
cat.eat/();  // Inherited from Animal        cat.meow(); // Specific to Cat    }}
```



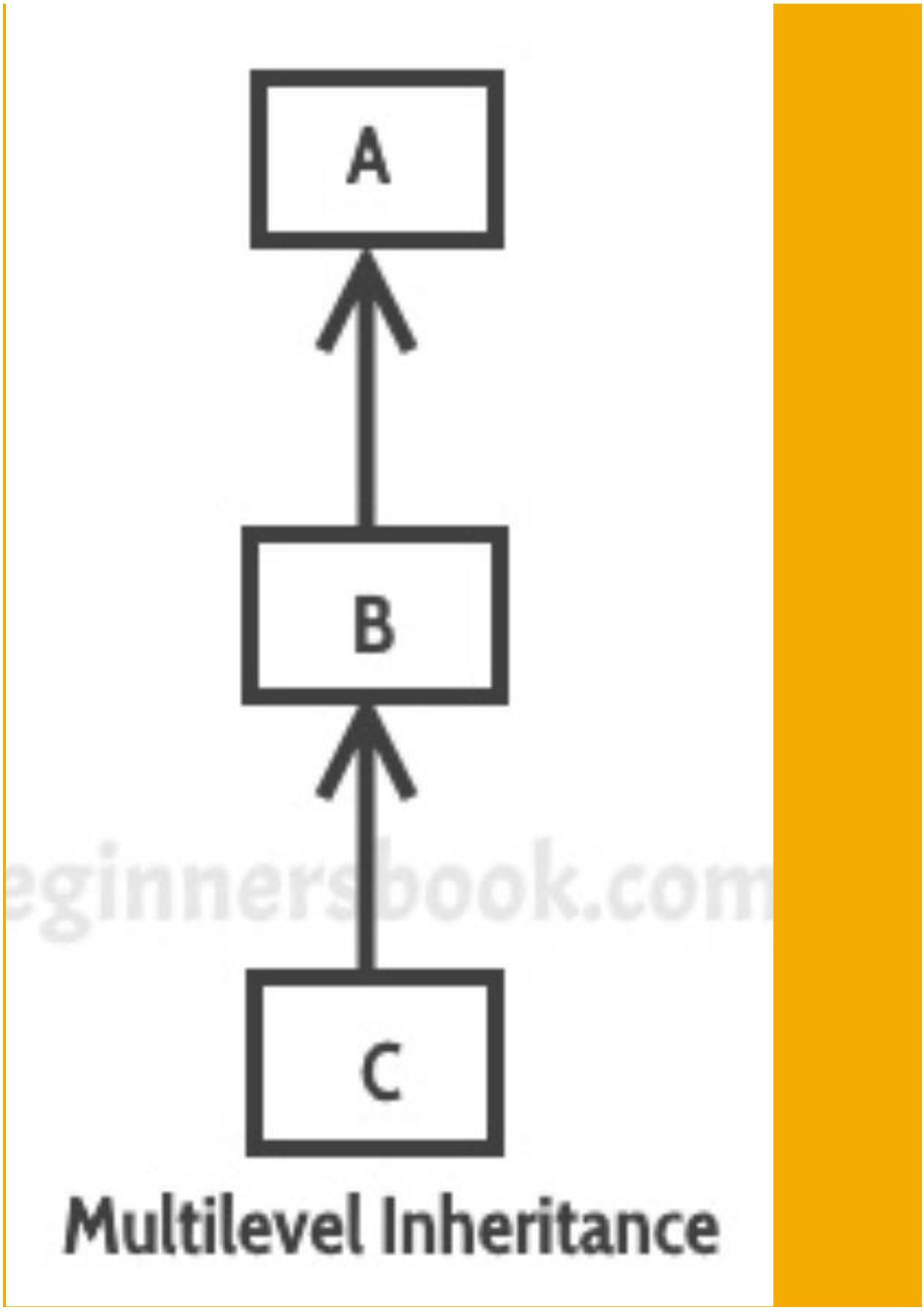
Multilevel inheritance

1. When a class extends a class, which extends anther class then this is called multilevel inheritance.

2. For example class C extends class B and class B extends class A then this type of inheritance is known as Multilevel inheritance.

```
class Car{
   public Car()
   {
System.out.println("Class Car");
   }
   public void vehicleType()
   {
System.out.println("Vehicle Type: Car");
   }
}
class Maruti extends Car{
   public Maruti()
   {
System.out.println("Class Maruti");
```

```
   }
   public void brand()
   {
System.out.println("Brand: Maruti");
   }
   public void speed()
   {
System.out.println("Max: 90Kmph");
   }
}
public class Maruti800 extends Maruti{

   public Maruti800()
   {
System.out.println("Maruti Model: 800");
   }
   public void speed()
   {
System.out.println("Max: 80Kmph");
   }
   public static void main(String args[])
   {
 Maruti800 obj=new Maruti800();
 obj.vehicleType();
 obj.brand();
 obj.speed();
   }
}
```


A
B
C
Multilevel Inheritance

Hybrid Inheritance

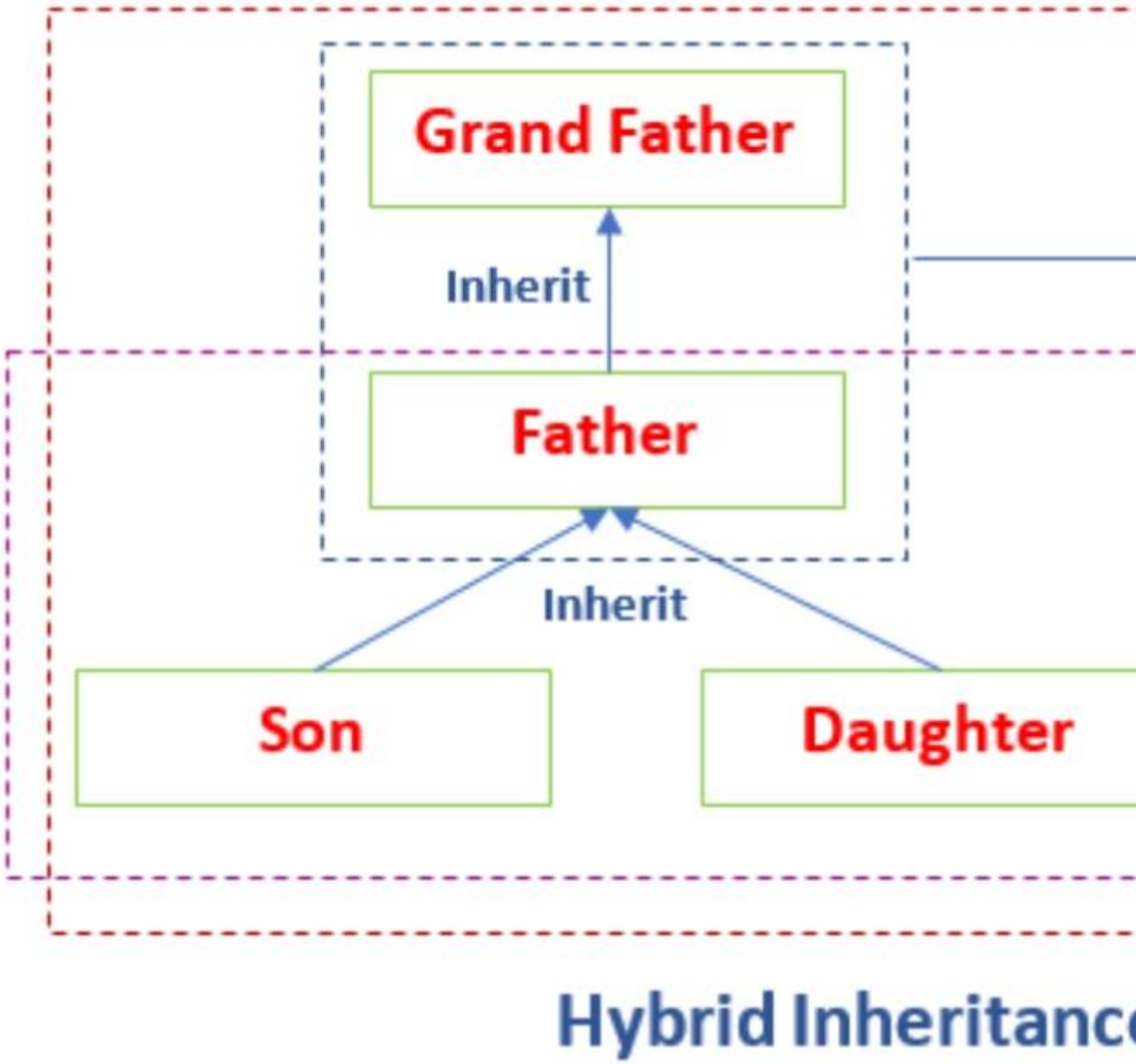
1. In general, the meaning of hybrid (mixture) is made of more than one thing.

2. In Java, the hybrid inheritance is the composition of two or more types of inheritance.

3. The main purpose of using hybrid inheritance is to modularize the code into well-defined classes.

4. It also provides the code reusability. 5. The hybrid inheritance can be achieved by using the following combinations:

1. Single and Multiple Inheritance (not supported but can be achieved through interface).
2. Multilevel and Hierarchical Inheritance.
3. Hierarchical and Single Inheritance.
4. Multiple and Multilevel Inheritance.



Need of inheritance in java

1. It allows you to create a new class by extending an existing class, thereby, inheriting its properties and behaviors.

2. This promotes code reuse and helps avoid redundancy.

3. Inheritance allows you to organize your code hierarchically, making it easier to understand and maintain.

4. It provides a mechanism for extending the functionality of existing classes without modifying them.

5. By creating subclasses, you can add new features or modify existing ones while keeping the original class intact.

6. Inheritance enables polymorphism, which allows objects of different classes to be treated as objects of a common superclass.

7. Inheritance helps in creating abstract classes and interfaces, which define a blueprint for other classes.

Interfaces

1. Interfaces allow you to define a set of methods without providing any implementation details.

2. This allows you to create a blueprint for classes to follow without specifying how they achieve the desired functionality.

3. Interfaces define a contract that classes must adhere to if they implement the interface.

4. Unlike classes, Java allows interfaces to support multiple inheritance.

5. Interfaces enable polymorphism, allowing objects of different classes to be treated interchangeably if they implement the same interface.

6.Interfaces can also contain constants, which are implicitly public, static, and final.

7. These constants provide a way to define and enforce standard values across multiple classes.

Super keyword

1. You can use super to access members (methods, fields, or constructors) of the superclass from within a subclass.

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

class Dog extends Animal {
    void eat() {
        super.eat/(); // Calls the eat() method of the superclass
        System.out.println("Dog is eating");
    }
}
```

2. In a subclass constructor, you can use super() to explicitly call a constructor of the superclass.

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

class Dog extends Animal {
    Dog() {
        super(); // Calls the constructor of the superclass (Animal)
        System.out.println("Dog constructor");
    }
}
```

3. You can use super multiple times to access members of ancestor classes as well.

```
class Animal {
    void sleep() {
        System.out.println("Animal is sleeping");
    }
}

class Dog extends Animal {
```

```
    void sleep() {
        super.sleep(); // Calls the sleep() method of the superclass
(Animal)
        System.out.println("Dog is sleeping");
    }
}

class Puppy extends Dog {
    void sleep() {
        super.sleep(); // Calls the sleep() method of Animal (ancestor
class)
        System.out.println("Puppy is sleeping");
    }
}
```

Unit 4 (Wrapper Classes, Arrays and Strings)

Wrapper class

1. In Java, a wrapper class is a class that wraps around a primitive data type and provides methods to manipulate, convert, and work with that data type as an object.
2. Examples include Integer, Double, Boolean, etc.
3. They allow primitives to be used in situations where objects are required, like collections or generics.
4. The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.

```
public class WrapperExample {
    public static void main(String[] args) {
        // Using Integer wrapper class
        Integer num1 = new Integer(10); // wrapping int 10
        Integer num2 = new Integer("20"); // wrapping string "20"
        // Performing operations using wrapper class methods
        int sum = num1.intValue() + num2.intValue(); // converting
Integer objects to int
        System.out.println("Sum: " + sum);
        // Autoboxing and unboxing
        Integer num3 = 30; // autoboxing: converting int to Integer
implicitly
        int product = num2 * num3; // unboxing: converting Integer to
int implicitly
        System.out.println("Product: " + product);
    }
}
```

In this example, we're using the Integer wrapper class to wrap primitive integers and perform operations using methods provided by the wrapper class. We also demonstrate autoboxing and unboxing, where Java automatically converts between primitive types and their corresponding wrapper classes.

Need of Wrapper class

1. They convert primitive data types into objects.

2. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).

3. The classes in java.util package handles only objects and hence wrapper classes help in this case also.

4. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.

5. An object is needed to support synchronization in multithreading.

Wrapper classes

Primitive type	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

Java.lang.Number Class

1. In java, while working with numbers, we use primitive data types.

2. But, Java also provides various numeric wrapper sub classes under the abstract class Number present in java.lang package.

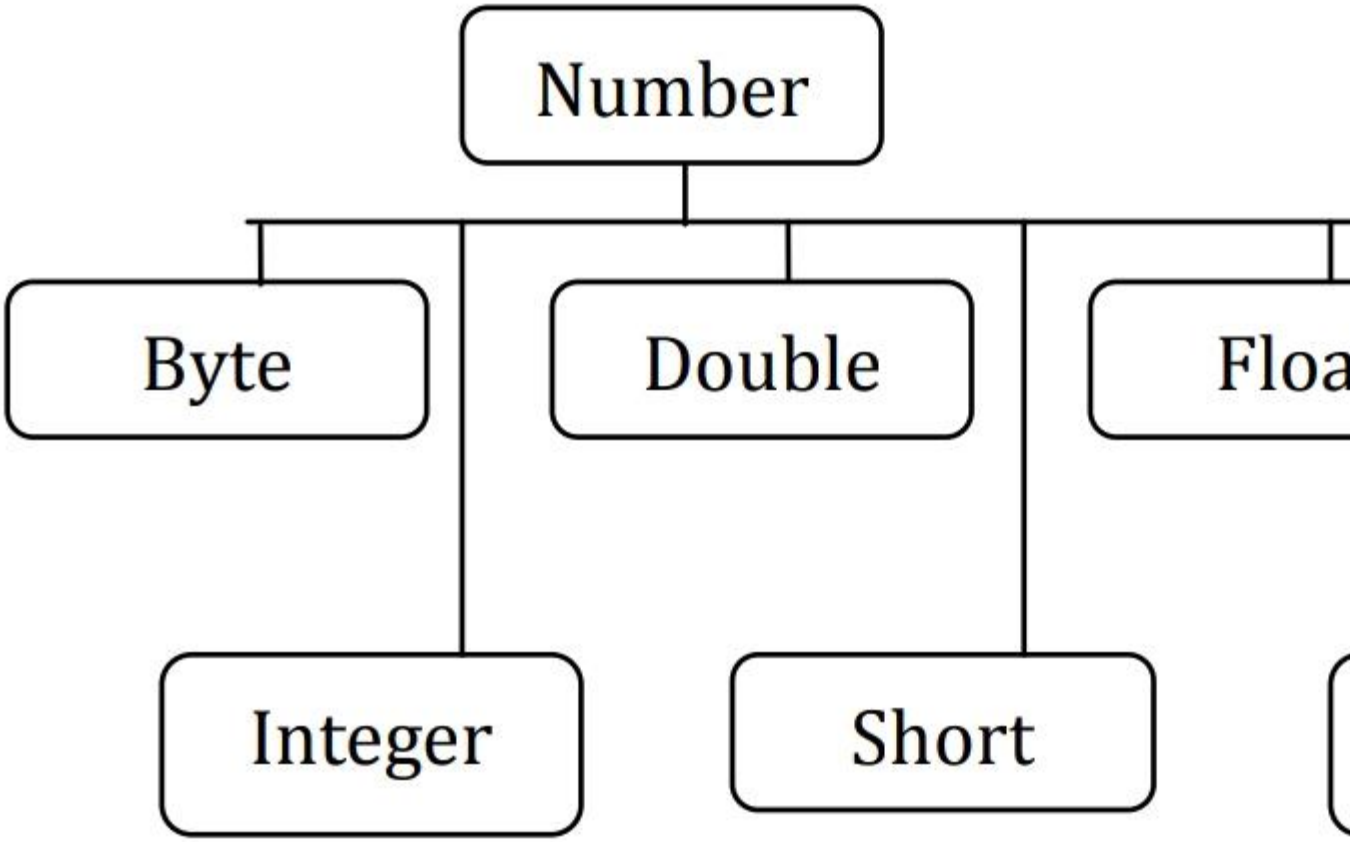
3. The Number class have mainly six sub-classes under it.

4. All these sub-classes define some useful methods which are used regularly while dealing with numbers.

5. These all derived classes can “wrap” the primitive data type in its corresponding object.

6. Often, the wrapping is done by the compiler.

7. Incase if we use a primitive values at the place where an object is expected, the compiler boxes the primitive values in its wrapper class for you and vice versa.



Methods of Number Class

Sr.No.	Method & Description
1	**xxxValue()**:It Converts the value of *this* Number object to th
2	**compareTo()**:It is used to Compares *this* Number object to t
3	**equals()**:It willDetermines whether *this* number object is eq
4	**valueOf()**: It will Returns an Integer object holding the value
5	**toString()**: Returns a String object representing the value of
6	**parseInt()**: This method is used to get the primitive data typ
7	**abs()**: This method willReturns the absolute value of the arg
8	**ceil()**: This methodReturns the smallest integer that is greate Returned as a double.
9	**floor()**: Returns the largest integer that is less than or equal t double.
10	**rint()**:Returns the integer that is closest in value to the argun
11	**round()**: Returns the closest long or int, as indicated by the r argument.
12	**min()**: This method used toReturns the smaller of the two arg
13	**max()**:It willReturns the larger of the two arguments.
14	**exp()**: Returns the base of the natural logarithms, e, to the p

Methods of Integer Class

Sr.No.	Method & Description
1	**toString () :** Returns the string corresponding to the int value.
2	**valueOf() :** returns the Integer object initialised with the value
3	**valueOf(String val,int radix):** Another overloaded function to new Integer(Integer.parseInteger(val,radix))
4	**valueOf(String val)** Another overloaded function which provides function similar t new Integer(Integer.parseInt(val,10))
5	**getInteger() :** returns the Integer object representing the v system property or null if it does not exist.
6	**decode () :** returns a Integer object holding the decoded value
7	**rotateLeft() :** Returns a primitive int by rotating the bits l complement form of the value given.
8	**rotateRight() :** Returns a primitive int by rotating the bits rig complement form of the value given.
9	**byteValue() :** returns a byte value corresponding to this Intege
10	**shortValue() :** returns a short value corresponding to this Inte
11	**floatValue() :** returns a float value corresponding to this Intege
12	**intValue() :** returns a value corresponding to this Integer Obje
13	**doubleValue() :** returns a double value corresponding to this I

Methods of Float Class

3	**parseFloat() :** returns float value by parsing the string. Diff primitive float value and valueOf() return Float object.
4	**byteValue() :** returns a byte value corresponding to this Floa
5	**shortValue() :** returns a short value corresponding to this Flo
6	**intValue() :** returns a int value corresponding to this Float O
7	**longValue() :** returns a long value corresponding to this Floa
8	**doubleValue() :** returns a double value corresponding to this
9	**floatValue() :** returns a float value corresponding to this Floa
10	**hashCode() :** returns the hashcode corresponding to this Floa
11	**isNaN() :** returns true if the float object in consideration is no
12	**isInfinite() :** returns true if the float object in consideration is
13	**equals() :** Used to compare the equality of two Float objects.
14	**compareTo() :** Used to compare two Float objects for numer
15	**compare() :** Used to compare two primitive float values for r
16	**toHexString() :** Returns the hexadecimal representation of th
17	**IntBitsToFloat() :** Returns the float value corresponding argument. It does reverse work of the previous two methods.

Methods of Byte Class

Sr.No.	Method & Description
1	**toString():** Returns the string corresponding to the byte value.
2	**valueOf() :** returns the Byte object initialised with the value pro
3	**parseByte() :** returns byte value by parsing the string.
4	**decode() :** returns a Byte object holding the decoded value of st
5	**byteValue() :** returns a byte value corresponding to this Byte O
6	**shortValue() :** returns a short value corresponding to this Byte
7	**intValue() :** returns a int value corresponding to this Byte Obje
8	**longValue() :** returns a long value corresponding to this Byte O
9	**doubleValue() :** returns a double value corresponding to this B
10	**floatValue() :** returns a float value corresponding to this Byte O
11	**hashCode() :** returns the hashcode corresponding to this Byte O
12	**equals() :** Used to compare the equality of two Byte objects.
13	**compareTo() :** Used to compare two Byte objects for numerica
14	**compare() :** Used to compare two primitive byte values for nu

Methods of Short Class

Sr.No.	Method & Description
1	**toString():** Returns the string corresponding to the Short val
2	**valueOf() :** returns the Short object initialised with the value
3	**parseShort() :** returns short value by parsing the string.
4	**decode() :** returns a Short object holding the decoded value
5	**byteValue() :** returns a byte value corresponding to this shor
6	**shortValue() :** returns a short value corresponding to this Sh
7	**intValue() :** returns a int value corresponding to this Short C
8	**longValue() :** returns a long value corresponding to this Sho
9	**doubleValue() :** returns a double value corresponding to this
10	**floatValue() :** returns a float value corresponding to this Sho
11	**hashCode() :** returns the hash code corresponding to this Sh

Methods of Long Class

Sr.No.	Method & Description
1	**toString():** Returns the string corresponding to the Long value.
2	**valueOf() :** returns the Long object initialised with the valu
3	**parse Long () :** returns Long value by parsing the string.
4	**decode() :** returns a Long object holding the decoded value
5	**byteValue() :** returns a byte value corresponding to this L
6	**shortValue() :** returns a short value corresponding to this I
7	**intValue() :** returns a int value corresponding to this Long
8	**longValue() :** returns a long value corresponding to this L
9	**doubleValue() :** returns a double value corresponding to th
10	**floatValue() :** returns a float value corresponding to this L
11	**hashCode() :** returns the hash code corresponding to this

Methods of Double Class

Sr.No.	Method & Description
1	**toString():** Returns the string corresponding to the Double valu
2	**valueOf() :** returns the Double object initialised with the value
3	**parse**Double() **:** returns Double value by parsing the string.
4	**decode() :** returns a Double object holding the decoded value o
5	**byteValue() :** returns a byte value corresponding to this Double
6	**shortValue() :** returns a short value corresponding to this Doub
7	**intValue() :** returns a int value corresponding to this Double O
8	**longValue() :** returns a long value corresponding to this Double
9	**doubleValue() :** returns a double value corresponding to this D
10	**floatValue() :** returns a float value corresponding to this Doubl
11	**hashCode() :** returns the hash code corresponding to this Doub

Methods of Character Class

Sr.No.	Method & Description
1	**toString(char ch)** : This method returns a String class ob character value(ch)
2	**char toLowerCase(char ch)** : It returns the lowercase of the s
3	**char toUpperCase(char ch)** : This method returns the up value(ch).
4	**boolean isLowerCase(char ch)** : It determines whether the sp lowercase or not.
5	**boolean isUpperCase(char ch)** : This method determine value(ch) is uppercase or not.
6	**char charValue()**: This method returns the value of this Chara
7	**static int compare(char x, char y)**: This method compares two
8	**int compareTo(Character anotherCharacter)**: This method numerically.
9	**static int digit(char ch, int radix)**: This method returns the nu in the specified radix.
10	**boolean equals(Object obj)**: This method compares this objec

Typecasting

Typecasting in Java is the process of converting a value from one data type to another. There are two types of typecasting:

1. Implicit Typecasting (Widening Conversion):

- Automatic conversion of data types by the compiler when there is no loss of information.

- For example, converting an integer to a float or a smaller integer type to a larger one.

-`byte` -> `short` -> `char` -> `int` -> `long` -> `float` -> `double`

2. Explicit Typecasting (Narrowing Conversion):

- Manual conversion of data types where there might be loss of information.

- It is done by the programmer using casting operators.

- For example, converting a float to an integer or a larger integer type to a smaller one.

-`double` -> `float` -> `long` -> `int` -> `char` -> `short` -> `byte`

Here's an example of explicit typecasting in Java:

```
public class TypecastingExample {
    public static void main(String[] args) {
        // Implicit typecasting (Widening Conversion)
        int numInt = 10;
        double numDouble = numInt; // int to double
        System.out.println("Implicit Typecasting (Widening
Conversion):");
        System.out.println("Integer: " + numInt);
        System.out.println("Double: " + numDouble);
        // Explicit typecasting (Narrowing Conversion)
        double doubleNum = 20.56;
        int intNum = (int) doubleNum; // double to int
        System.out.println("\nExplicit Typecasting (Narrowing
Conversion):");
        System.out.println("Double: " + doubleNum);
        System.out.println("Integer: " + intNum);
    }
}
```

In this example, we demonstrate both implicit and explicit typecasting. Implicit typecasting occurs when an integer is automatically converted to a double. Explicit typecasting occurs when a double is manually casted to an integer, which may result in loss of decimal information.

Autoboxing and unboxing

1. Autoboxing:

- It is the automatic conversion of primitive types to their corresponding wrapper class objects.

- It happens when a primitive type is assigned to a wrapper class object.

- For example, assigning an int to an Integer or a double to a Double.

2. Unboxing:

- It is the automatic conversion of wrapper class objects to their corresponding primitive types.

- It happens when a wrapper class object is used in a context where a primitive type is expected.

- For example, using an Integer object in an arithmetic operation where an int is expected.

Here's an example demonstrating autoboxing and unboxing in Java:

```
public class AutoboxingUnboxingExample {
    public static void main(String[] args) {
        // Autoboxing: primitive to wrapper object
        Integer numInteger = 10; // Autoboxing int to Integer
        Double numDouble = 20.5; // Autoboxing double to Double
        System.out.println("Autoboxing:");
        System.out.println("Integer: " + numInteger);
        System.out.println("Double: " + numDouble);
        // Unboxing: wrapper object to primitive
        int intNum = numInteger; // Unboxing Integer to int
        double doubleNum = numDouble; // Unboxing Double to
double
        System.out.println("\nUnboxing:");
        System.out.println("Integer: " + intNum);
        System.out.println("Double: " + doubleNum);
    }
}
```

In this example, autoboxing occurs when primitive values (int and double) are assigned to their corresponding wrapper classes (Integer and Double). Unboxing occurs when those wrapper class objects are used in contexts where primitive types are expected (int and double).

Array

1. Java array is an object which contains elements of a similar data type.

2. The elements of an array

are stored in a contiguous also called as sequential memory location.

3. It is a data structure where

we store similar types of elements.

4. In java array allows to store only a fixed set of elements.

5. In Java all arrays are dynamically allocated.

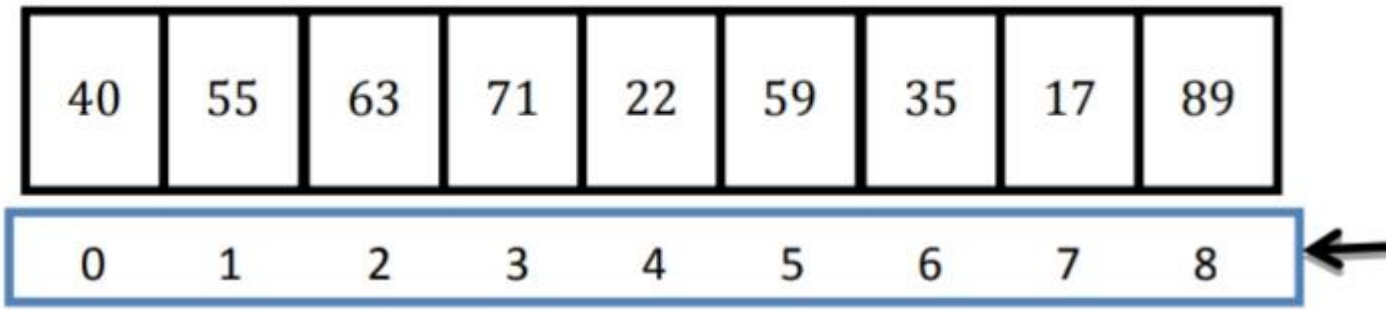
6. Since arrays are objects in Java, we can find their length using member length.

7. A Java array variable can also be declared like other variables with [] after the data type.

8. The variables in the array are ordered and each have an index beginning from 0.

9. Java array can be also be used as a static field, a local variable or a method parameter.

10. The size of an array must be specified by an int value and not long or short.



Array Length: 9
First Index: 0
Last Index: 8

Single dimensional array

1. A single-dimensional array in Java is a linear collection of elements of the same data type, accessed by a single index.

2. It's declared with a specified size and can store elements of primitive or reference types.

3. Accessing elements in a single-dimensional array is done using their index, starting from 0 to the length of the array minus one.

```
public class SingleDimensionalArrayExample {
    public static void main(String[] args) {
        // Declaration and initialization of an integer array
        int[] numbers = new int[5]; // Declares an array of size 5

        // Initialization of array elements
        numbers[0] = 10;
        numbers[1] = 20;
        numbers[2] = 30;
        numbers[3] = 40;
        numbers[4] = 50;

        // Accessing array elements
```

```
        System.out.println("Element at index 0: " + numbers[0]);
        System.out.println("Element at index 2: " + numbers[2]);
        System.out.println("Element at index 4: " + numbers[4]);
    }
}
```

Multidimensional Java

1. A multidimensional array in Java is an array of arrays, where each element of the array can itself be an array.

2. This allows for the creation of tables or matrices with multiple rows and columns.

3. In Java, you can create two-dimensional, three-dimensional, or even higher-dimensional arrays.

```
public class MultidimensionalArrayExample {
    public static void main(String[] args) {
        // Declaration and initialization of a 2D array
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };
        // Accessing elements in a 2D array
        System.out.println("Element at row 0, column 0: " +
matrix[0][0]);
        System.out.println("Element at row 1, column 2: " +
matrix[1][2]);
        System.out.println("Element at row 2, column 1: " +
matrix[2][1]);
    }
}
```

---

array of Objects

1. The array of Objects the name itself suggests that it stores an array of objects.

2. Unlike the traditional array stores values like String, integer, Boolean, etc an Array of Objects stores objects that mean objects are stored as elements of an array.

3. Note that when we say Array of Objects it is not the object itself that is stored in the array but the reference of the object.

```
public class ObjectArrayExample {
    public static void main(String[] args) {
        // Declaration and initialization of an array of objects
        Person[] people = new Person[3];
        // Creating objects and assigning them to array elements
        people[0] = new Person("Alice", 25);
        people[1] = new Person("Bob", 30);
        people[2] = new Person("Charlie", 28);
        // Accessing and printing object properties
        for (Person person : people) {
            System.out.println("Name: " + person.getName() + ", Age: "
+ person.getAge());
        }
    }
}
class Person {
    private String name;
    private int age;
    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // Getters
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
}
```

Unit 5 (String Handling and Exception Handling)

String

1. In Java, a string is a sequence of characters. It's an object of the `String` class, which is part of the `java.lang` package.
2.Strings in Java are immutable, meaning once created, their values cannot be changed.
3.strings are immutable in Java, any method that appears to modify a string actually returns a new string object with the desired modifications. The original string remains unchanged.
4. Here's how you can declare and initialize a string in Java:

```
String myString = "Hello, World!";
```

String Methods

- `length()`: Returns the length of the string.
- `charAt(int index)`: Returns the character at the specified index.
- `substring(int beginIndex)`: Returns a substring starting from the specified index.
- `substring(int beginIndex, int endIndex)`: Returns a substring between the specified indices.
- `toUpperCase()`: Converts the string to uppercase.
- `toLowerCase()`: Converts the string to lowercase.
- `equals(Object obj)`: Compares the content of two strings.
- `equalsIgnoreCase(String anotherString)`: Compares two strings ignoring case.
- `indexOf(String str)`: Returns the index of the first occurrence of the specified substring.
- `concat(String str)`: Concatenates the specified string to the end of this string

---

StringBuffer

1. StringBuffer is a class in Java that represents a mutable sequence of characters.

2. It provides an alternative to the immutable String class, allowing you to modify the contents of a string without creating a new object every time.

3. A string buffer is similar to String in functionalities, but can be modified.

4. It offers particular sequence of characters, and content of the sequence can be changed through certain method calls.

5. They are safe for use by multiple threads.

6. Every string buffer has a capacity.

7. Overall, if you need to perform multiple modifications to a string, or if you need to access a string from multiple threads, using StringBuffer can be more efficient and safer than using regular String objects.

StringBuffer Class methods

Sr.No.	Method & Description
1	**StringBuffer append(boolean b):**This method appends the string representation of the boolean argument to the sequence
2	**StringBuffer append(char c):** This method appends the string representation of the char argument to this sequence.
3	**StringBuffer append(char[] str):** This method appends the string representation of the char array argument to this sequence.
4	**StringBuffer** append(char[] str, int offset, int len): This method appends the string representation of a subarray of the char array argument to this sequence.
5	**StringBuffer append(CharSequence s):** This method appends the specified CharSequence to this sequence.
6	**StringBuffer append(CharSequence s, int start, int end):** This method appends a subsequence of the specified CharSequence to this sequence.
7	**StringBuffer append(double d):** This method appends the string representation of the double argument to this sequence.
8	**StringBuffer append(float f):** This method appends the string representation of the float argument to this sequence.
9	**StringBuffer append(int i):** This method appends the string representation of the int argument to this sequence.
10	**StringBuffer append(long lng):** This method appends the string representation of the long argument to this sequence.
11	**StringBuffer append(Object obj):** This method appends the string representation of the Object argument.

StringBuilder

1. StringBuilder in Java represents a mutable sequence of characters.

2. The function of StringBuilder is very much similar to the StringBuffer class, as both of them provide an alternative to String Class by making a mutable sequence of characters.

3. The StringBuilder class provides no guarantee of synchronization whereas the StringBuffer class does.

4. Where possible, it is recommended that this class be used in preference to StringBuffer as it will be faster under most implementations.

5. Instances of StringBuilder are not safe for use by multiple threads.

6. If such synchronization is required then it is recommended that StringBuffer be used.

7. String Builder is not thread-safe and high in performance compared to String buffer.

---

methods of StringBuilder class

1. **Appending content:**
   - `append(String str)`: Appends the specified string to the sequence.
   - `append(Object obj)`: Appends the string representation of the specified object.
   - `append(char c)`: Appends the specified character to the sequence.
2. **Inserting content:**
   - `insert(int offset, String str)`: Inserts the specified string into the sequence at the specified position.
   - `insert(int offset, char c)`: Inserts the specified character into the sequence at the specified position.
3. **Deleting content:**
   - `delete(int start, int end)`: Deletes the characters in a substring of this sequence.
   - `deleteCharAt(int index)`: Removes the character at the specified position in this sequence.
4. **Replacing content:**
   - `replace(int start, int end, String str)`: Replaces the characters in a substring of this sequence with characters in the specified string.
5. **Other methods:**
   - `reverse()`: Causes this character sequence to be replaced by the reverse of the sequence.
   - `length()`: Returns the length (number of characters) of the sequence.
6. **Converting to String:**
   - `toString()`: Converts the `StringBuilder` object into a `String`.

StringTokenizer

1. StringTokenizer class in Java is used to breakdown a string into small sequences of charecters called as token.

2. This class is a legacy class that is retained for compatibility reasons although its use is discouraged in new code.

3. The methods of StringTokenizers do not differentiate between numbers, identifiers, and quoted strings.

4. This class methods do not even recognize and skip comments.

5. A StringTokenizer object internally maintains a current position within the string to be tokenized.

6. Some operations advance this current position past the characters processed.

7. To create the StringTokenizer object substring of the string is used and then the token is returned.

StringTokenizer Class methods

Sr.No.	Method & Description
1	**int countTokens():** This method calculates the number of times that this tokenizer's nextToken method can be called before it generates an exception.
2	**boolean hasMoreElements():** This method returns the same value as the hasMoreTokens method.
3	**boolean hasMoreTokens():** This method tests if there are more tokens available from this tokenizer's string.
4	**Object nextElement():** This method returns the same value as the nextToken method, except that its declared return value is Object rather than String.
5	**String nextToken():** This method returns the next token from this string tokenizer.
6	**String nextToken(String delim):** This method returns the next token in this string tokenizer's string.

StringJoiner

1. StringJoiner is a class in java.util package is used to construct a sequence of characters(strings) separated by a delimiter and optionally starting with a supplied prefix and ending with a given suffix.

2. Though this can also be done with the help of the StringBuilder class to append delimiter after each string, StringJoiner provides an easy way to do that without much code to write.

3. It's particularly useful for joining elements of a collection (such as a list) into a single string with a specified delimiter.

StringJoiner Methods

1. **Constructor:**
   - `StringJoiner(CharSequence delimiter)`: Creates a `StringJoiner` with the given delimiter.
2. **Adding elements:**
   - `StringJoiner add(CharSequence newElement)`: Adds a new element to the `StringJoiner`.
   - `StringJoiner merge(StringJoiner other)`: Merges another `StringJoiner` into this one.
3. **Setting prefix and suffix:**
   - `StringJoiner setEmptyValue(CharSequence emptyValue)`: Sets the value to return if the `StringJoiner` is empty.
   - `StringJoiner setPrefix(CharSequence prefix)`: Sets the sequence of characters to be used as the prefix.
   - `StringJoiner setSuffix(CharSequence suffix)`: Sets the sequence of characters to be used as the suffix.
4. **Getting the result:**
   - `String toString()`: Returns the current value of the `StringJoiner` as a `String`.

Difference between String and StringBuffer

No.	String	Strin
1)	The String class is immutable.	The St
2)	String is slow and consumes more memory when we concatenate too many strings because every time it creates new instance.	String we co
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	String metho
4)	String class is slower while performing concatenation operation.	String conca
5)	String class uses String constant pool.	String

Difference Between StringBuffer and StringBuilder

No.	StringBuffer	StringBuilder
1)	StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-sy* threads can call the met
2)	StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more eff*
3)	StringBuffer was introduced in Java 1.0	StringBuilder was introd

---

Exception

In Java, an exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. When an exceptional condition arises, an object representing that condition, called an "exception object," is created and thrown within the method that encountered the error. This is known as "throwing an exception."

Exceptions can occur for various reasons, such as:

- Runtime Errors: These occur during the execution of the program, such as division by zero or accessing an array element out of its bounds.
- Logic Errors: These occur due to mistakes in the program's logic, leading to unexpected behavior or incorrect results.
- External Factors: These include errors like file not found, network connection issues, or database errors, which are beyond the control of the program.

Exception sub type

1. Runtime Exception or Unchecked Exception: These exceptions are need not be handled before compilation of the source code. Complier doesn't check whether the exceptions are handled before compilation or not that's why they are called as Unchecked exceptions.

2. Checked Exception: Checked exception are checked at compile time and application should handle these exceptions before the compilation of program.

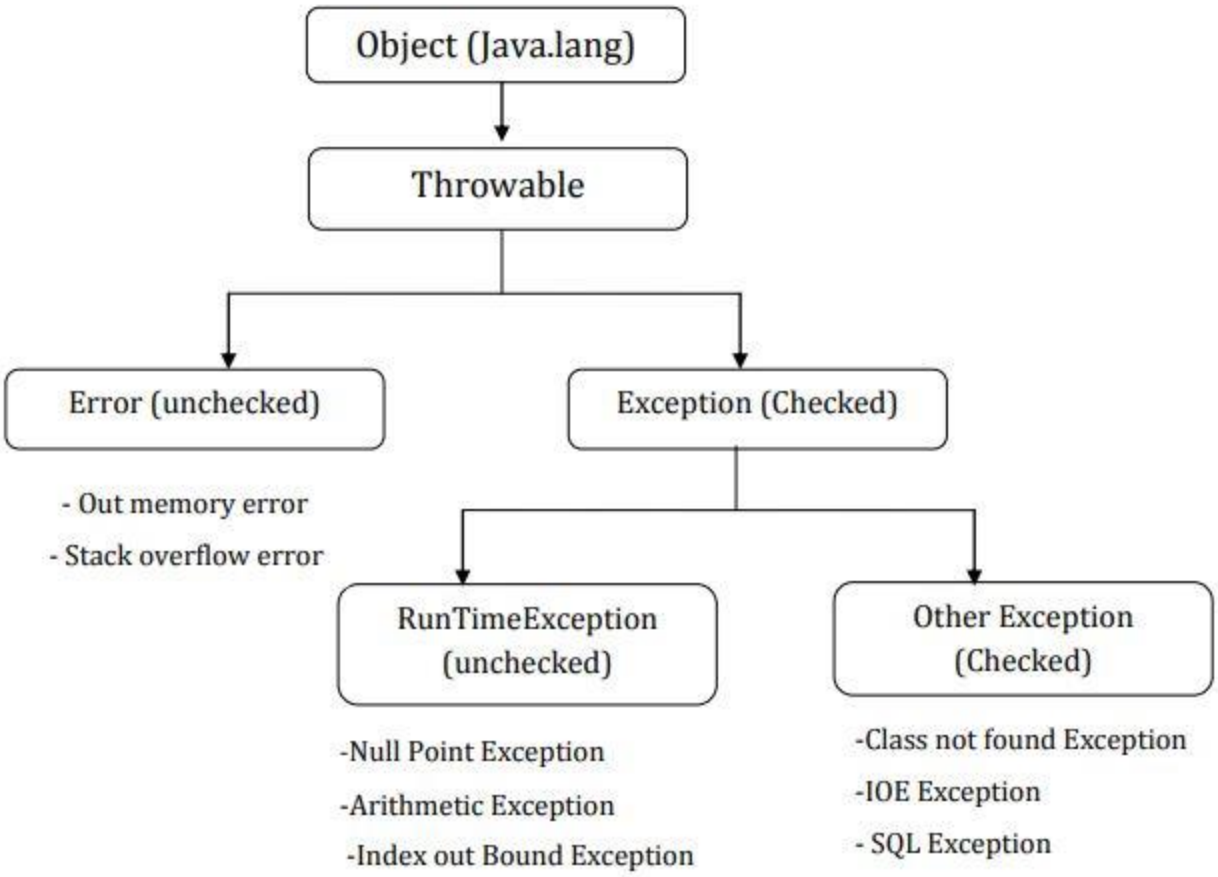
3. If these exceptions are not handled it will not allow you to even compile the source code.

4.That's why they are called as Checked Exception.

5. User-defined custom exception: We can also create our own exception. Here are a few rules:

- Throwable must be the superclass of all exceptions.
- One needs to extend the RunTimeException class to to create a RuntimeException.
- One needs to extend the Exception class to create the checked exception.

Exception Hierarchy



Exceptions Methods

Sr. No.	Method & Description
1	**public String getMessage():** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	**public Throwable getCause():** Returns the cause of the exception as represented by a Throwable object.
3	**public String toString():** Returns the name of the class concatenated with the result of getMessage().
4	**public void printStackTrace():** Prints the result of toString() along with the stack trace to System.err, the error output stream.
5	**public StackTraceElement [] getStackTrace():** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	**public Throwable fillInStackTrace():** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Try Block

1. The "try" is a keyword which is used to mention a block in which exception code or risky code can be placed.

2. It means we write the The code which may cause an exception (risky code) in try block.

3. If exception occur then suddenly compiler will terminate the exception of try block and will start the execution of catch or finally block.

4. The try block must be used with the combination of either catch or finally. In simple words, try block alone can not be used.

5. Here's the syntax of a `try` block:

```
try {
    // Code that may throw an exception
} catch (ExceptionType1 ex1) {
    // Handle ExceptionType1
} catch (ExceptionType2 ex2) {
    // Handle ExceptionType2
} finally {
    // Optional finally block
}
```

Catch

1. The "catch" block is used to handle the exception.

2. It must be used with and after try block.

3. In simple words catch block alone can not be used.

4. If the exception is occurred in try block then only catch block will be executed.

5. In catch block we write a code to handle the exception or to describe more information about exception to user.

6. It may or may not be used with finally block.

---

finally

The `finally` block in Java is used to execute a block of code regardless of whether an exception is thrown or caught within the `try` block. It ensures that certain cleanup or resource release tasks are performed, even if an exception occurs.

The syntax for a `finally` block is as follows:

```
try {
    // Code that may throw an exception
} catch (ExceptionType ex) {
    // Exception handling code
} finally {
    // Code that always executes, regardless of whether an exception
occurred or not
    // Usually used for cleanup or resource release tasks
}
```

Throw

The "throw" keyword is used to create and throw an Exception. It means user can manually createor generate the exception using throw keywords.

Throws

1.The "throws" keyword is used to announce exceptions.

2.It doesn't throw an exception.
3. It indicates that there may occur an exception in the method.
4. It informs the compiler that the specified methods may throws the exception, in that case user dosent have to handle the exception that will be handled by the compiler.
5. The throws will always be used along with method signature.

---

exception handling example

```
import java.util.Scanner;
```

```
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
            System.out.print("Enter a number: ");
            int num1 = scanner.nextInt();

            System.out.print("Enter another number: ");
            int num2 = scanner.nextInt();

            int result = num1 / num2;
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: Division by zero is not
allowed.");
        } catch (Exception e) {
            System.out.println("An error occurred: " +
e.getMessage());
        } finally {
            // Close resources or perform cleanup operations
            scanner.close();
        }
    }
```

Unit 6 (Package and Deferred Implementation)

Package

1. A Package is a collection of related classes.
2. It helps arrange your classes into a folder structure
and make it simple to trace and use them.
3. More importantly, it helps improve re-usability.
4. A package in Java allows compressing a group of classes, interfaces, enumerations, annotations, and sub-packages.
5. In simple word, you can consider of java packages as being similar to different folders on your computer.
6. When software is created in any programming language, it can be consisting of hundreds or even thousands of individual classes.
7. It makes efficient to mantain things structured by placing related
classes and interfaces into packages.
8. Each package arranges its classes and interfaces into a separate namespace, or name group and each one has its unique name.

Built-in Package

The package consist of a large number of classes which are a part of Java API. Some of the commonly used built-in packages are:

1) java.lang: This is a default package, i.e this package is automatically imported and contains language support classes such as classes which defines primitive data types, math operations etc.

2) *java.io/*: This package contains classes that support various input / output operations.
3) java.util: This package contains utility classes which can be used to implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
4) java.applet: This package contains classes for creating Applets.
5) java.awt: This package contains classes for implementing the components for graphical user interfaces like button, frames, menus etc.
6) *java.net/*: This package contains classes for supporting networking operations.

User-defined packages

```
//Name/ of the package must be same as the directory
// under which this file is saved
package myPackage;
public class MyClass
{
 public void getNames(String s)
 {
 System.out.println(s);
 }
}
```

Now we can use the MyClass class in our program by importing 'MyClass' class from 'names' myPackage

```
import myPackage.MyClass;
public class PrintName
{
 public static void main(String args[])
 {
 // Initializing the String variable
 // with a value
 String name = "Welcome to My Package";
 // Creating an instance of class MyClass in
 // the package.
 MyClass obj = new MyClass();
 obj.getNames(name);
 }
}
```

Access Protection in Java Packages

1. Java offers diffrent access control mechanism and its access specifiers.

2. Packages in Java add another layer to access control.

3. The collection of classes and packages together called as data encapsulation.

4. The packages in java act as containers for classes, interfaces and other subordinate packages, where as classes in package act as containers for data and code.

5. This relationship between packages and classes inJava, packages offer four categories of visibility for class

members:

- Sub-classes in the same package.
- Non-subclasses in the same package.
- Sub-classes in different packages.
- Classes that are neither in the same package nor sub-classes.

	Private	No Modifier	Pro
Same Class	Yes	Yes	
Same Package Subclasses	No	Yes	
Same Package Non-Subclasses	No	Yes	
Different Packages Subclasses	No	No	
Different Packages Non- Subclasses	No	No	

---

abstract class

1. Generally, an abstract class in Java is a template that stores the data members and methods that we use in a program.

2. Abstraction in Java keeps the user from viewing complex code implementations and provides the user with necessary information.

3. We cannot instantiate the abstract class in Java directly.

4. Instead, we can subclass the abstract class. When we use an abstract class as a subclass, the abstract class method implementation becomes available to all of its parent classes.

5. To declare an abstract class, we use the access modifier first, then the "abstract" keyword, and the class name shown below.

//Syntax/:

```
<Access_Modifier> abstract class <Class_Name> {

//Data_Members/;

//Statements/;

//Methods/;

}
```

Abstraction using interface

1. An interface in Java is a specification of method prototypes.

2. Whenever you need to guide the programmer or, make a contract specifying how the methods and fields of a type should be you can define an interface.

3. To create an object of this type you need to implement this interface, provide a body for all the abstract methods of the interface and obtain the object of the implementing class.

4. The user who want to use the methods of the interface, he only knows the classes that implements this interface and their methods, information about the implementation is completely hidden from the user, thus achieving 100% abstraction.

```
interface <interface_name>{

    // declare constant fields
    // declare methods that abstract
```

```
    // by default.
}
```

Difference between abstract class and interface.

Abstract class	In
Abstract class can have abstract and non-abstract methods.	Interface can have Since Java 8, it can methods also.
Abstract class doesn't support multiple inheritance.	Interface supports mul
Abstract class can have final, non-final, static and non-static variables.	Interface has only stati
The implementation of interface can be provided by abstract class.	The implementation o provided by interface.
To declare abstract classabstract keyword is used.	To declare interface t used.
An abstract class can implement multiple Java interfaces and can extend another Java class.	An interface canonlyin interface by extending
An abstract class can be extended using keyword "extends".	An interface can be keyword "implements"
A Java abstract class can have class members with different access specifiers like private, protected, etc.	In Java interface all default.
Example: public abstract class Shape{ public abstract void draw();	Example: public interface Drawa void draw();

Generalization

1. Generalization is the process, which extracts the shared features from two or more classes, and combines the extracted features into a generalized superclass.

2. Shared features can be attributes,

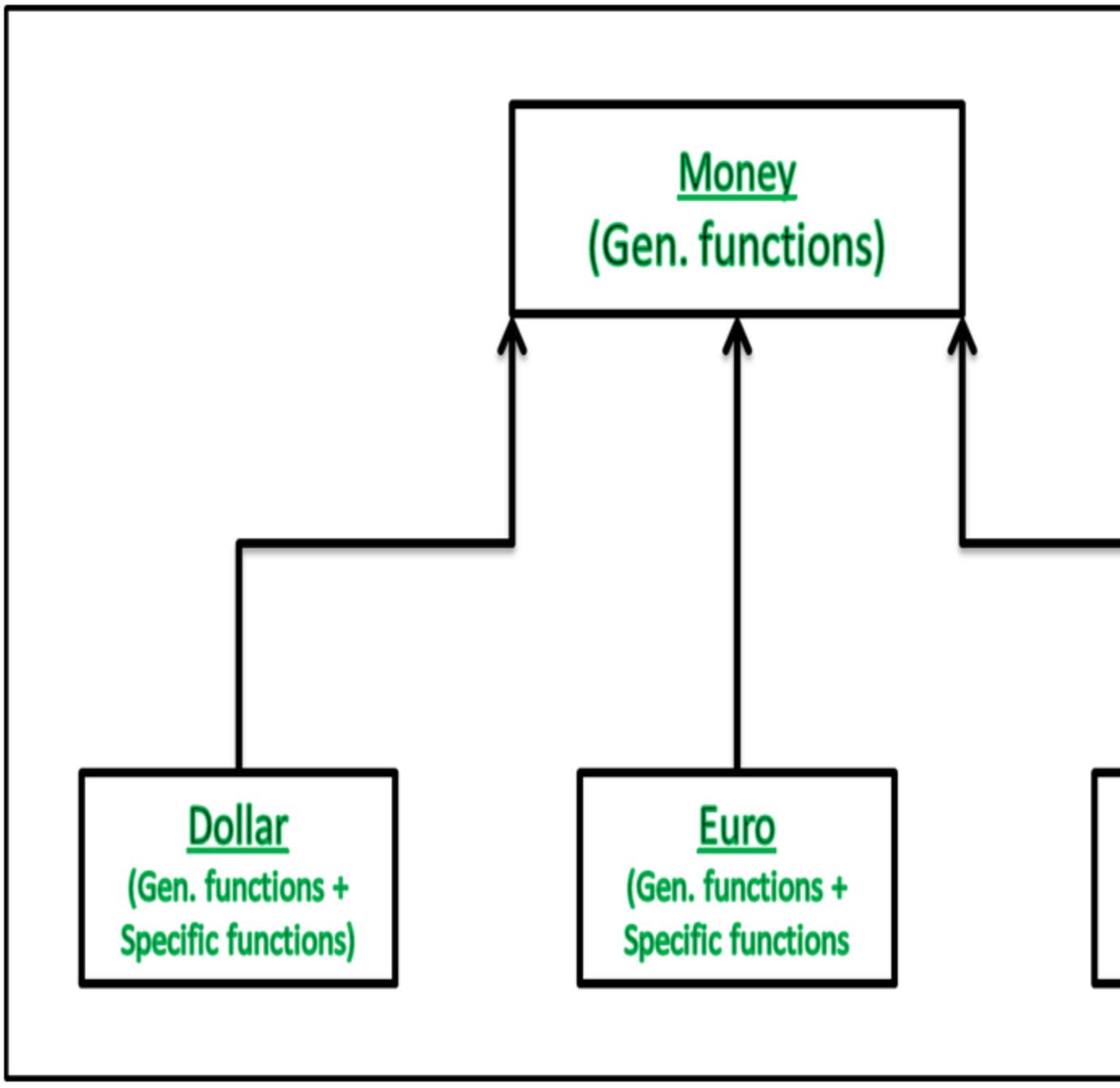
associations, or methods.

3. The process of Generalization is the bottom-up process of abstraction, in which we club the differences among entities according to the common feature and generalize them into a single superclass.

4. The original entities are then becomes subclasses of it.

5. The process of converting a subclass type into a superclass type is called

"Generalization" because we are making the subclass to become more general and its scope is widening.



Specialization

1. Converting a super class type into a sub class type is called ‘Specialization‘.

2. Here, we are coming down from more general form to a specific form and hence the scope is narrowed.

3. Hence, this is called narrowing or down-casting.

4. Narrowing is not safe because the classes will become more and more specific thus giving rise to more and more doubts.

5. For example if we say Vehicle is a Car we need a proof.

6.Thus, In this case, Java compiler specifically asks for the casting.

7.This is called explicit casting.

8. Example: To show when Narrowing is not allowed:

```
class Father {

    public void work()

    {

        System.out.println("Earning Father");

    }
}



class Son extends Father {

    public void play()

    {

        System.out.println("Enjoying son");

    }
}
```

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

        try {
            // son is a sub class reference
            Son son;

            // new operator returns a superclass reference
            // which is narrowed using casting
            // and stored in son variable

            // This will throw exception
            son = (Son) new Father();

            // Through a narrowed reference of the superclass
            // we can neither access superclass method
            // and nor the subclass methods

            // Below lines will show
            // an error when uncommented
            // son.work/();
            // son.play/();
```

```
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

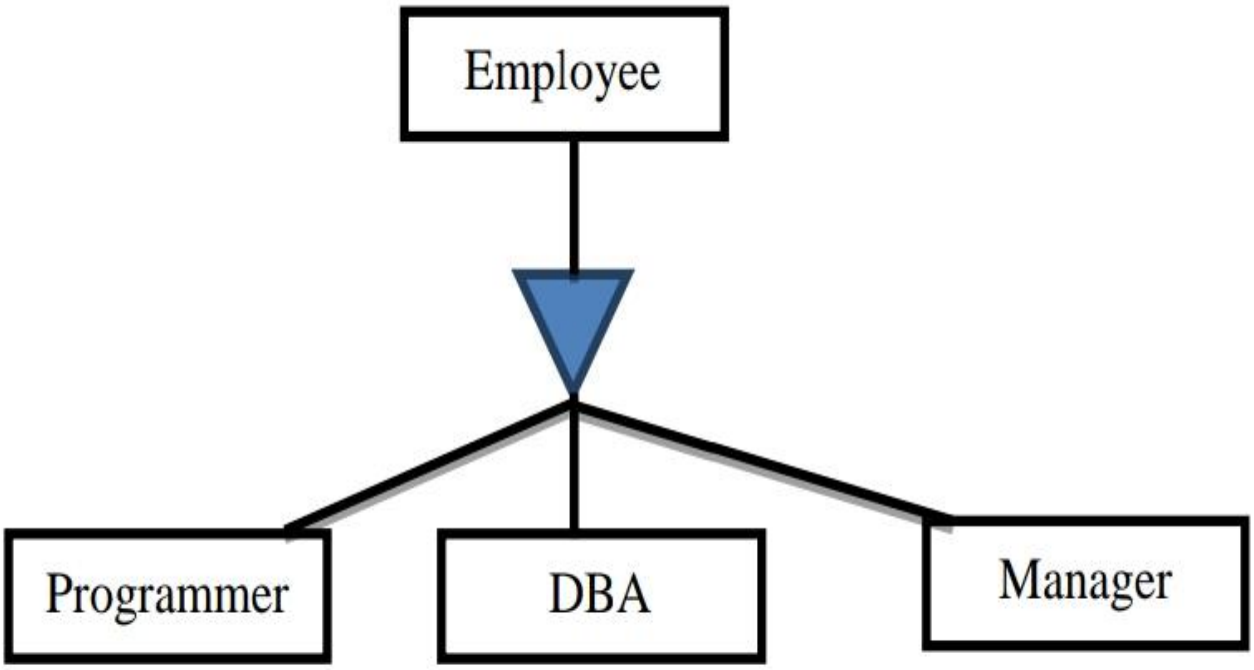


Figure.6.4: Specialization Example

Unit 7 (Java I/O)

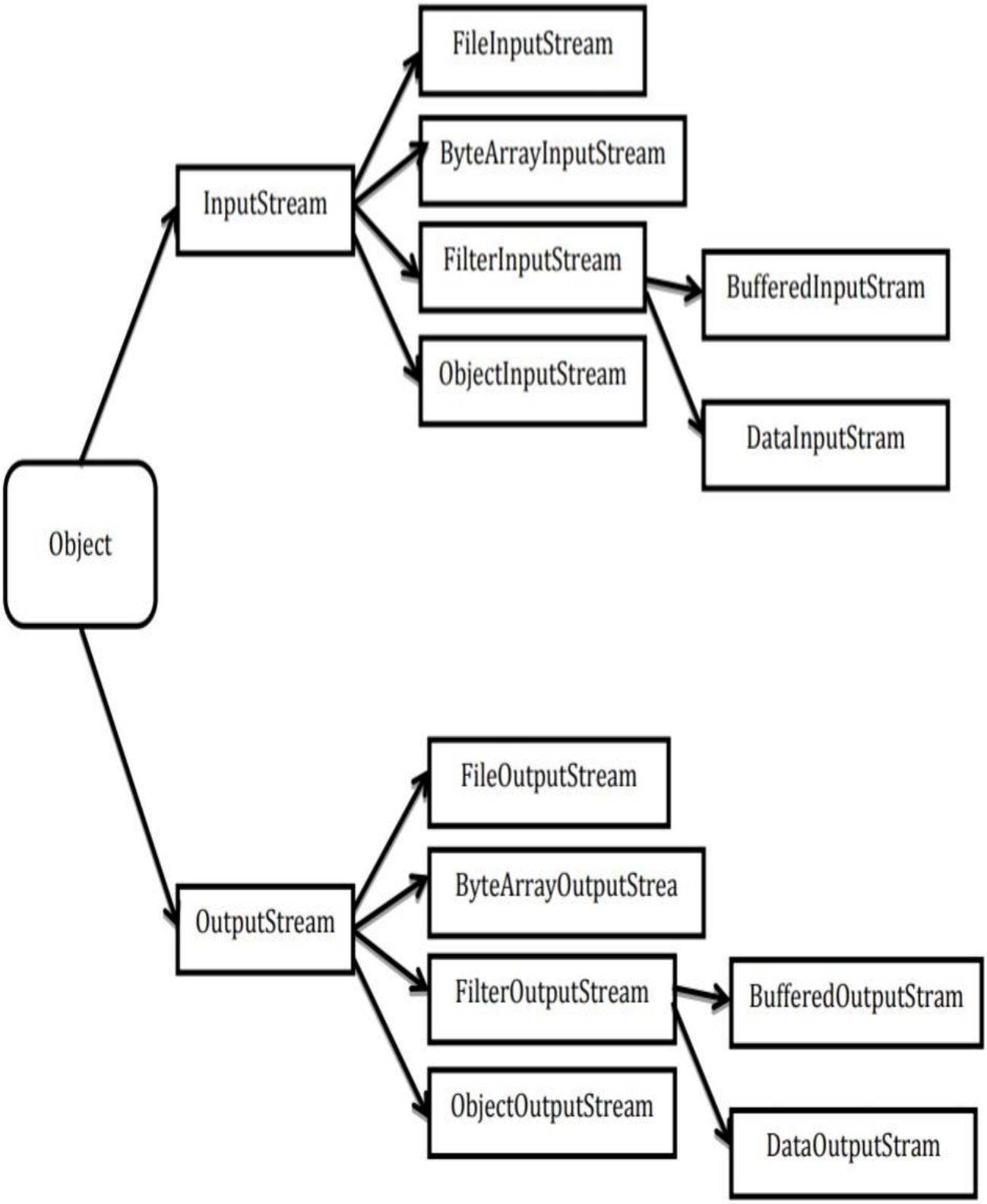
File Class

1. Java File class is Java's representation of a file or directory path name.
2. Because file and directory names have different formats on different platforms, a simple string is not adequate to name them.
3. Java File class contains several methods for working with the pathname, deleting and renaming files, creating new directories, listing the contents of a directory, and determining several common attributes of files and directories.
4. It is an abstract representation of files and directory path names.
5. we should create the File class object by passing the file name or directory name to it.
6. A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing.
7. Instances of the File class are immutable.

File Class Methods

Sr. No	Methods & Descriptions
1	boolean canExecute() : Tests whether the application can execute the file denoted by this abstract pathname.
2	boolean canRead() : Tests whether the application can read the file denoted by this abstract pathname.
3	boolean canWrite() : Tests whether the application can modify the file denoted by this abstract pathname.
4	int compareTo(File pathname) : Compares two abstract pathnames lexicographically.
5	boolean createNewFile() : Atomically creates a new, empty file named by this abstract pathname .
6	static File createTempFile(String prefix, String suffix) : Creates an empty file in the default temporary-file directory. boolean delete() : Deletes the file or directory denoted by this abstract pathname.
7	boolean equals(Object obj) : Tests this abstract pathname for equality with the given object.
8	boolean exists() : Tests whether the file or directory denoted by this abstract pathname exists.
9	String getAbsolutePath() : Returns the absolute pathname string of this abstract pathname.
10	long getFreeSpace() : Returns the number of unallocated bytes in the partition .
11	String getName() : Returns the name of the file or directory denoted by this abstract

Hierarchy of Object


Object
InputStream
FileInputStream
ByteArrayInputStream
FilterInputStream
ObjectInputStream
BufferedInputStram
DataInputStram
OutputStream
FileOutputStream
ByteArrayOutputStrea
FilterOutputStream
ObjectOutputStream
BufferedOutputStram
DataOutputStram

FileOutputStream

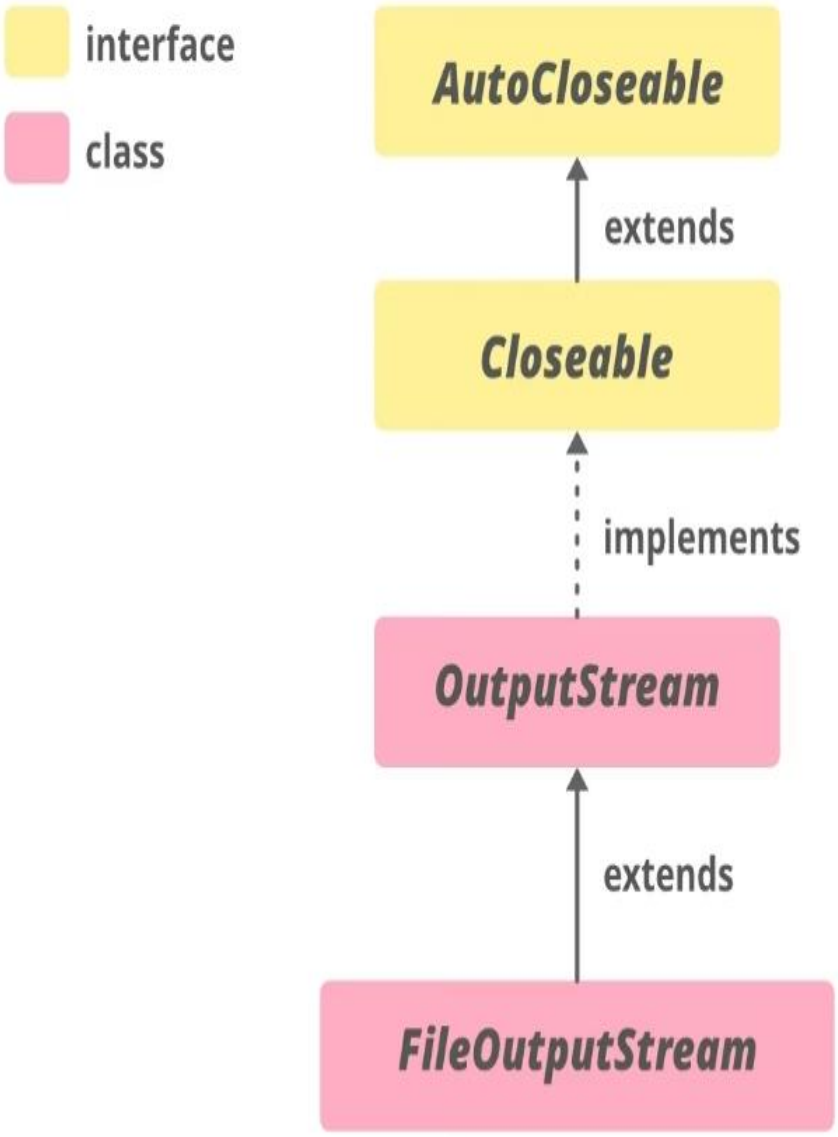
1. FileOutputStream is an outputstream for writing data/streams of raw bytes to file or storing data to file.

2. FileOutputStream is a subclass of OutputStream.

3. To write primitive values into a file, we use FileOutputStream class.

4. For writing byte-oriented and character-oriented data, we can use FileOutputStream but for writing character-oriented data, FileWriter is more preferred.

Hierarchy of FileOutputStream



FileInputStream

1. A FileInputStream is a Java class that enables reading bytes from a file in a file system.

2. It's part of the Java I/O (Input/Output) API and is used to read data from files as a stream of bytes.

3. With FileInputStream, you can read data from a file byte by byte or in chunks, depending on your requirements.

BufferedWriter Class

The BufferedWriter class in Java is used for writing characters to a stream efficiently by buffering them in memory before writing to the underlying output stream.

Here's a brief overview of the BufferedWriter class:

1. Buffering: BufferedWriter stores data in an internal buffer before writing it to the underlying output stream.

2. Character Streams: BufferedWriter works with character streams, which means it's typically used to write text data to a file or another output stream.

3. Constructor: BufferedWriter requires an instance of another Writer (such as FileWriter or another BufferedWriter) as its argument in its constructor.

4. Flushing: BufferedWriter automatically flushes its buffer to the underlying stream under certain conditions, such as when the buffer is full.

5. Close Method: It's essential to close a BufferedWriter after use to release system resources and ensure that any buffered data is properly flushed and written to the output stream.

Methods of BufferedWriterClass

Method	Description
void newLine()	This method adds a new line by writing a line separator.
void write(int c)	This method is used to write a single character.
void write(char[] cbuf, int off, int len)	This method writes a portion of an array of characters.
void write(String s, int off, int len)	This method is used to write a portion of a string.
void flush()	This methodused to flushes the input stream.
void close()	This method is used to closes the input stream

Java BufferedReader Class

The BufferedReader class in Java is used for reading text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

Here's a brief overview of the BufferedReader class:

1. Efficiency: BufferedReader provides efficient reading of characters, arrays, and lines from a character-input stream.

2. Read Operations: It offers various methods for reading text data, including reading a single character, reading into a character array, or reading an entire line.

3. Buffering: BufferedReader uses an internal buffer to reduce the number of reads from the underlying input stream.

4. Convenience: BufferedReader simplifies the reading process by providing methods like readLine(), which reads a line of text, and read(), which reads a single character.

5. Supports Marking: BufferedReader supports the marking of a position in the input stream and later resetting the stream to that position.

BufferedReader class methods

Method	Description
int read()	This methodis used for reading a single character.
int read(char[] cbuf, int off, int len)	This method is used for reading characters into a portion of an array.
boolean markSupported()	This methodtest the input stream support for the mark and reset method.
String readLine()	This methodused for reading a line of text.
boolean ready()	This method tests whether the input stream is ready to be read.
long skip(long n)	This methodskip the characters specified by position number passed to it.
void reset()	This method repositions the stream at a position the mark method was last called on this input stream.
void mark(int readAheadLimit)	To mark the present position in a stream mark method is used.
void close()	This method is used to releases any of the system resources associated with the stream and also closes the input stream.

Serialization and Deserialization

1. Serialization is a mechanism of converting the state of an object into a byte stream.

2. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory.

3. This mechanism is used to persist the object.

4. The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

5. To make a Java object serializable we implement the *java.io/*.Serializable interface.
6. The ObjectOutputStream class contains writeObject() method for serializing an Object.
7. If a parent class has implemented Serializable interface then child class doesn't need to implement it but vice-versa is not true.
8. Only non-static data members are saved via Serialization process.

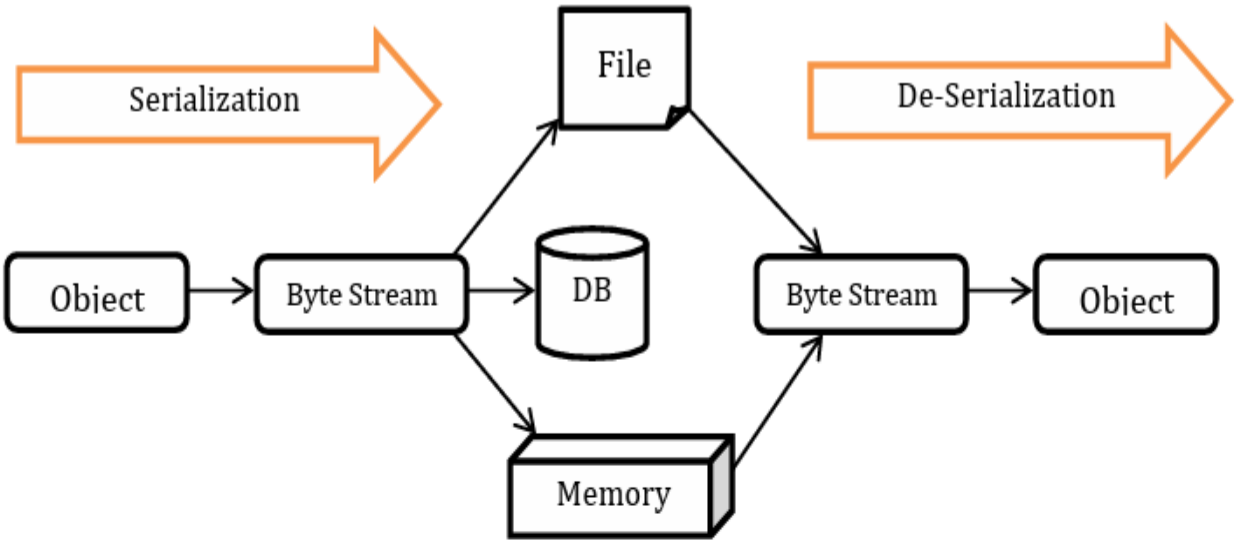


Fig. 7.3(a): Serialization and Deserialization

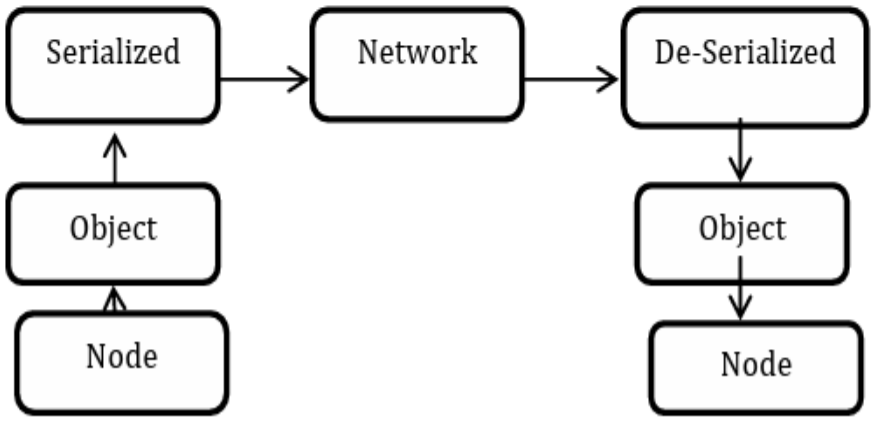


Fig. 7.3(b): Serialization and Deserialization

Scanner class

1. Scanner class in Java is part of java.util package.

2. In Java the java.util.Scanner class is one of the method to read input from the keyboard.

3. The Java Scanner class divide the input into different tokens with the help of a delimiter which is whitespace by default.

4. To read and parse various primitive values various methods are offered by it.

5. The Java Scanner class uses a regular expression to parse text for strings and primitive types.

6. Java Scanner is the easiest way to accept input in Java.

7. Scanner in Java support us to get input from the user in primitive types such as int, long, double, byte, float, short, etc.

8. The Java Scanner class has Object class as Super class and this class also inherits the interfaces Iterator and Closeable by implementing them.

Scanner Class Methods

24)	byte nextByte()	This method is used toacceptthe next token of the input as a byte.
25)	double nextDouble()	This method is used toacceptthe next token of the input as a double.
26)	float nextFloat()	This method is used to acceptthe next token of the input as a float.
27)	int nextInt()	This method is used toacceptthe next token of the input as an Int.
28)	String nextLine()	It is used to get the input string that was skipped of the Scanner object.
29)	long nextLong()	This method is used toacceptthe next token of the input as a long.
30)	short nextShort()	This method is used to accept the next token of the input as a short.
31)	int radix()	This method is used to get the default radix of the Scanner use.
32)	void remove()	when remove operation is not supported by this implementation of Iterator then this method can be used.
33)	Scanner reset()	This method is used to reset the Scanner which is in use.
34)	Scanner skip()	It skips input that matches the specified pattern, ignoring delimiters
35)	Stream<String>tokens()	It is used to get a stream of delimiter-separated tokens from the Scanner object which is in use.
36)	String toString()	It is used to get the string representation of Scanner using.
37)	Scanner useDelimiter()	It is used to set the delimiting pattern of the Scanner which is in use to the specified pattern.
38)	Scanner useLocale()	It is used to sets this scanner's locale object to the specified locale.

Unit 8 (Thread, Generics and Collection)

Threading

1. A thread is a process or task in execution.
2. The Java Virtual Machine allows an application to execute multiple threads concurrently.
3. Such as paint, calculator, Microsoft word or any process running on computer is a thread.
4. Every thread has a priority.
5. Threads with higher priority are executed in preference to threads with lower priority.
6. Each thread which is currently executing may or may not be marked as a daemon.
7. When a new Thread object is created, the priority of new thread is initially set equal to the priority of the creating thread.
8. A newly created thread is a daemon thread if and only if the creating thread is a daemon.

Thread creation by extending the Thread class

We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside the run() method. We create an object of our new class and call the start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

```
class MyThread extends Thread {
    public void run() {
        System.out.println("MyThread is running");
    }

    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start(); // This starts the execution of the thread
    }
```

Implementing the Runnable Interface

Implementing the `Runnable` interface is another way to create a thread in Java. This approach separates the thread's behavior from the thread's execution mechanism, offering more flexibility.

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("MyRunnable is running");
    }

    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread thread = new Thread(myRunnable);
        thread.start(); // This starts the execution of the thread
    }
}
```

Lifecycle of a thread

1. New Thread: When a new thread is created, it is in the new state. Its code is yet to be run and hasn't started to execute.

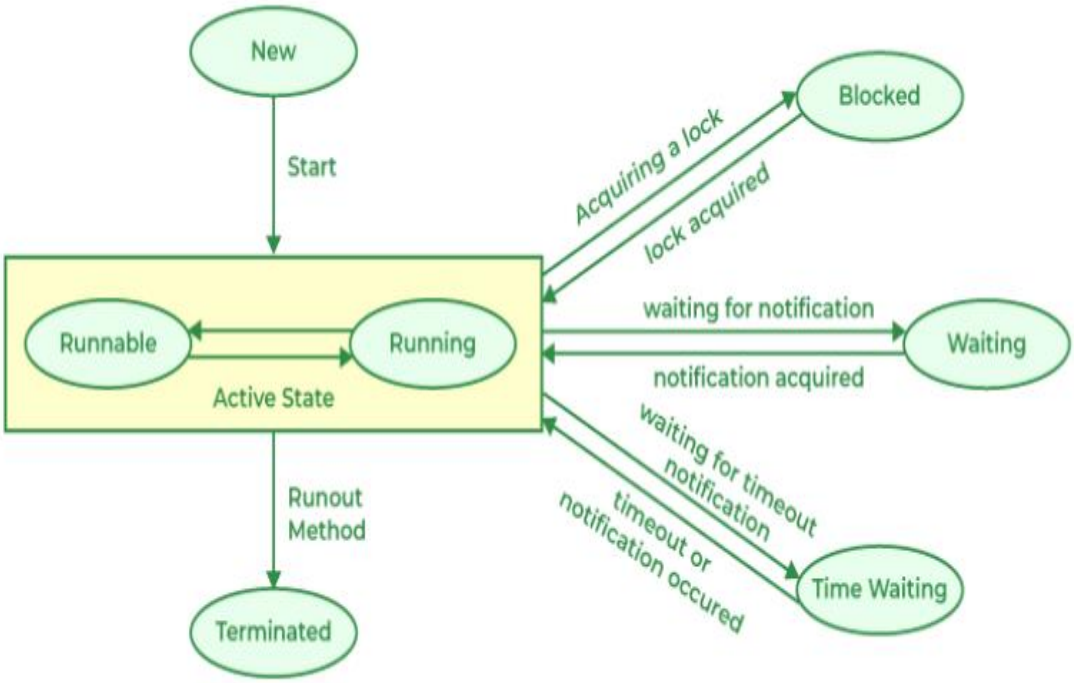
2. Runnable State: In this state, a thread might actually be running or it might be ready to run at any instant of time.

3. Blocked: The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread.

4. Waiting state: The thread will be in waiting state when it calls wait() method or join() method.

5. Timed Waiting: A thread lies in this state until the timeout is completed or until a notification is received.

6. Terminated State: A thread terminates because either the code of the thread has been entirely executed by the program. Because it exits normally or Because there occurred some unusual erroneous event, like an unhandled exception.



Methods in Thread Class

Sr. No.	Method & Descriptions
1	**activeCount()**:This method returns an estimate of the number of active threads in the current thread's thread group and its subgroups
2	**checkAccess()**:This method determines if the currently running thread has permission to modify this thread
3	**clone()**:Throws CloneNotSupportedException as a Thread can not be meaningfully cloned
4	**currentThread()**:This method returns a reference to the currently executing thread object
5	**dumpStack()**:It will Prints a stack trace of the current thread to the standard error stream
6	**getAllStackTraces()**: This method returns a map of stack traces for all live threads
7	**getId()**: Returns the identifier of this Thread
8	**getName()**:Returns this thread's name
9	**getPriority()**:Returns this thread's priority
10	**getStackTrace()**:This method Returns an array of stack trace elements representing the stack dump of this thread
11	**getState()**:Returns the state of this thread
12	**getThreadGroup()**:This method Returns the thread group to which this thread belongs
13	**interrupt()**:Interrupts this thread
14	**interrupted()**:Tests whether the current thread has been interrupted
15	**isAlive()**:Tests if this thread is alive
16	**join()**: Waits for this thread to join

Thread Priority

1. NORM_PRIORITY: This constant represents the default priority assigned to threads in Java. Its value is typically set to 5.

2. MIN_PRIORITY: This constant represents the minimum priority that a thread can have. Its value is typically set to 1.

3. MAX_PRIORITY: This constant represents the maximum priority that a thread can have. Its value is typically set to 10.

4. The default priority for the main thread is always 5.

5. The default priority for all other threads depends on the priority of the parent thread.

6. If two threads have the same priority then we can't expect which thread will execute first.

7. It depends on the thread scheduler's algorithm.

8. If we are using thread priority for thread scheduling then we should always keep in mind that the underlying platform should provide support for scheduling based on thread priority.

Synchronization

1. Multi-threaded programs may often enter in to a situation where multiple threads try to access the same resources and finally produce invalid and unforeseen results.

2. So synchronization method is needed to make confirm that only one thread can access the resource at a given point of time.

3. Java provides steps and ensured method of creating threads and synchronizing their task by using synchronized blocks.

4. The synchronized keyword can be used in java to define Synchronized.

5. A synchronized block in Java is synchronized on some object.

6. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time.

7. All other threads struggling to enter the synchronized block are remains in waiting until the thread inside the synchronized block exits the block.

Java Generics

1. Java Generics is an advanced feature of java that can be implemented with methods and classes called as generic method and generic class respectively.

2. Generic methods and generic classes permit programmers to write a single method declaration, which can be used to specify a set of related methods, and write a single class declaration to specify, a set of related types of classes.

3. Generics in java also offer compile-time type safety, which permits programmers to catch invalid types at compile time.

4. Generics in Java are exactly analogous to templates in C++.

5. The Concept is to allow type (Integer, String, etc. and user defined types) to be a parameter to methods, classes and interfaces.

Advantage of Java Generics

1. Type-safety: It is a feature of generic that doesn‟t allow storing other objects; we can hold only a single type of objects in generics.

2. Type casting is not required: Typecasting of object is not required.

3. Compile-Time Checking: The type of generic parameter is checked at compile time so problem will not occur at runtime. To handle the problem at compile time rather than runtime is a good programming strategy.

4. Code Reusability: it allows code reusability as you can reuse the same code for different Types of data.

Generic Methods

1. Method that can be called with arguments of different types is called as generic method.

2. The parameters in generic method can be replaced by any types.

3. The compiler identifies and handles different method call appropriately, based on the types of the parameters passed to the generic method.

4. The generic method declarations have a argument type which is delimited in angle brackets (< and >) and method is precedes by return type (< E >).

5. parameter section contains one or more number of parameters with different types separated by commas.

6. The type parameters can be used to act as placeholders for the types of the arguments passed to the generic method as well and can also be used to declare the return type.

7. A generic method's body is declared just like that of any normal method.

Collections in Java

1. A collection in java, as name indicates, is an assembly or group of objects.

2. Java Collections framework is consist of the different interfaces and classes which can be used in implementation with different types of data structures as a collections such as lists, sets, maps, stacks and queues etc.

3. These inbuild collection classes solve lots of very common problems where we need to deal with group of homogeneous as well as heterogeneous objects.

4. The basic operations offered by collection are add, remove, update, sort, search and more complex algorithms.

5. These collection classes offer easy and very clear provision for all such operations using Collections APIs.

Collections Hierarchy

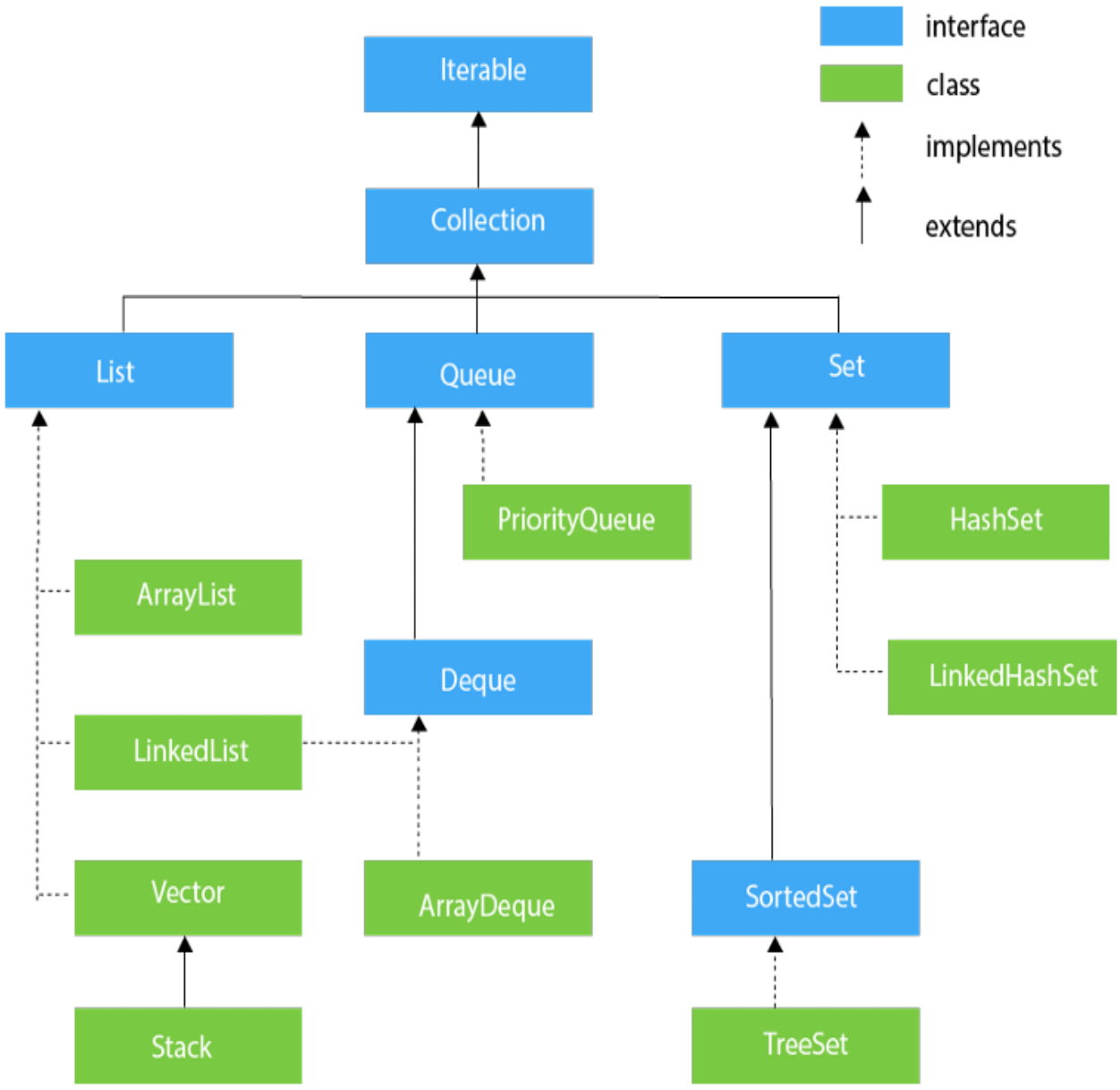


Fig.8.2: Java collections Hierarchy

List

1. In Java, a list is a data structure used to store a collection of elements.

2. The java.util.List interface is a part of the Java Collections Framework and defines operations for working with lists.

3. Some common implementations of the List interface include ArrayList, LinkedList, and Vector.

Here's a brief overview of each:

- ArrayList: This is a resizable array implementation of the List interface. It dynamically resizes itself when elements are added or removed.
- LinkedList: This implementation uses a doubly linked list to store elements. It provides efficient insertion and deletion operations, especially when elements are frequently added or removed from the beginning or middle of the list.
- Vector: This is a legacy implementation similar to ArrayList but is synchronized, meaning it's thread-safe.

```
import java.util.ArrayList;
import java.util.List;

public class ListExample {
    public static void main(String[] args) {
        // Creating an ArrayList
        List<String> myList = new ArrayList<>();

        // Adding elements to the list
        myList.add("Apple");
        myList.add("Banana");
        myList.add("Orange");

        // Accessing elements by index
        System.out.println("First element: " + myList.get(0));
```

```
        // Iterating through the list
        System.out.println("Elements:");
        for (String fruit : myList) {
            System.out.println(fruit);
        }

        // Removing an element
        myList.remove("Banana");
        System.out.println("After removing Banana:");
        for (String fruit : myList) {
            System.out.println(fruit);
        }
    }
}
```

Set

1. In Java, Set is an interface in the java.util package that represents a collection of unique elements.

2. It does not allow duplicate elements.

3. The Set interface is implemented by several classes in Java, with HashSet, TreeSet, and LinkedHashSet being the most commonly used implementations.

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();

        set.add("apple");
        set.add("banana");
        set.add("apple"); // Ignored, as it's a duplicate

        System.out.println(set); // Output: [banana, apple]
    }
}
```

Map

1. The Map interface stores the data in key-value pairs, where keys should be immutable.

2. A duplicate key is not allowed in Map; each key can map to at most one value.

3. The Map interface provides three different views to map contents, which are a map‟s contents to

be viewed as a set of keys, collection of values, or set of key-value mappings.

4. Some of the map operations, like the TreeMap class, make particular assurances as to their order; others, like the HashMap class, do not.

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();

        map.put("apple", 10);
        map.put("banana", 5);
        map.put("apple", 20); // Updates the value associated with
"apple"

        System.out.println(map.get("apple")); // Output: 20
        System.out.println(map.containsKey("banana")); // Output: true
    }
}
```

Stack

1. In Java, a stack is a data structure that follows the Last In, First Out (LIFO) principle, meaning that the last element added to the stack is the first one to be removed.

2. The Java standard library provides an implementation of the stack data structure through the java.util.Stack class.

3. However, it's generally recommended to use the Deque interface, specifically ArrayDeque or LinkedList, instead of Stack, as Stack extends Vector, which is a synchronized collection and is considered less efficient for most use cases.

```
import java.util.Deque;
import java.util.ArrayDeque;

public class Main {
    public static void main(String[] args) {
        // Creating a stack using Deque interface with ArrayDeque
implementation
        Deque<Integer> stack = new ArrayDeque<>();

        // Pushing elements onto the stack
        stack.push(10);
        stack.push(20);
        stack.push(30);

        // Peeking at the top element without removing it
        System.out.println("Top element of the stack: " +
stack.peek());

        // Popping elements from the stack
        System.out.println("Popping elements:");
        while (!stack.isEmpty()) {
            System.out.println(stack.pop());
        }
    }
}
```

Queue

1. A queue data structure is anticipated to contain the elements prior to processing by consumer thread(s).

2. Moreover basic operations offered by collections, queues offer supplementary insertion, extraction, and inspection operations.

3. Queues characteristically, order elements in a FIFO (first-in-first-out) manner, but which is not necessarily in every situation.

4. One of such situation or example is priority queue which order elements according to a supplied Comparator called as priority, or the elements‟ natural ordering.

```
import java.util.Queue;
import java.util.LinkedList;
```

```
public class Main {
    public static void main(String[] args) {
        // Creating a queue using LinkedList
        Queue<String> queue = new LinkedList<>();

        // Adding elements to the queue
        queue.offer("First");
        queue.offer("Second");
        queue.offer("Third");

        // Peeking at the front element without removing it
        System.out.println("Front element of the queue: " +
queue.peek());

        // Removing and printing elements from the queue
        System.out.println("Polling elements:");
        while (!queue.isEmpty()) {
            System.out.println(queue.poll());
        }
    }
}
```

Dequeue

1. A double ended queue is two ended queue that supports element insertion and removal at both ends.

2. When a deque is implemented as a queue, FIFO (First-In-First-Out) behavior results.

3. Deque can also be treated as a stack, LIFO (Last-In-First-Out) behavior results.

4. This interface should be used in preference to the legacy Stack class.

5. When elements are inserted and removed from the beginning of the deque it can be treated as a stack.

6. Some of the commonly used classes implementing this interface are ArrayDeque, ConcurrentLinkedDeque, LinkedBlockingDeque and LinkedList.

```
import java.util.Deque;
import java.util.ArrayDeque;
```

```
public class Main {
    public static void main(String[] args) {
        // Creating a deque using ArrayDeque
        Deque<Integer> deque = new ArrayDeque<>();

        // Adding elements to the front and back of the deque
        deque.offerFirst(10);
        deque.offerLast(20);
        deque.offerLast(30);

        // Peeking at the first and last elements without removing
them
        System.out.println("First element: " + deque.peekFirst());
        System.out.println("Last element: " + deque.peekLast());

        // Removing and printing elements from the front and back of
the deque
        System.out.println("Polling elements from the deque:");
        while (!deque.isEmpty()) {
            System.out.println(deque.pollFirst());
        }
    }
}
```

Linked List

1. The LinkedList class is a collection which can contain many objects of the same type, just like the ArrayList.

2. The LinkedList class has all of the same methods as the ArrayList class because they both implement the List interface.

3. However, while the ArrayList class and the LinkedList class can be used in the same way, they are built very differently.

4. The LinkedList stores its items in "containers." The list has a link to the first container and each container has a link to the next container in the list.

5. To add an element to the list, the element is placed into a new container and that container is linked to one of the other containers in the list.

Vector Class

1. The java.util.Vector class implements a growable array of objects.

2. Similar to an Array, it contains components that can be accessed using an integer index.

3. The Vectors is scalable that is size of a Vector can rise or shrink as desired to accommodate adding and removing items.

4. Each vector tries maintaining a capacity and a capacity Increment to maintain the memory optimization.

5. Unlike the new collection implementations, Vector is synchronized.

6. This class is a member of the Java Collections Framework.

Vector Methods

5	void addElement(E obj): This method adds the specified component to the end of this vector, increasing its size by one.
6	int capacity(): This method returns the current capacity of this vector.
7	void clear(): This method removes all of the elements from this vector.
8	clone clone(): This method returns a clone of this vector.
9	boolean contains(Object o): This method returns true if this vector contains the specified element.
10	boolean containsAll(Collection<?> c): This method returns true if this Vector contains all of the elements in the specified Collection.
11	void copyInto(Object[ ] anArray): This method copies the components of this vector into the specified array.
12	E elementAt(int index): This method returns the component at the specified index.
13	Enumeration<E> elements(): This method returns an enumeration of the components of this vector.
14	void ensureCapacity(int minCapacity): This method increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
15	boolean equals(Object o): This method compares the specified Object with this Vector for equality.
16	E firstElement(): This method returns the first component (the item at index 0) of this vector.
17	E get(int index): This method returns the element at the specified position in this Vector.
18	int hashCode(): This method returns the hash code value for this Vector.
19	int indexOf(Object o): This method returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element.
20	int indexOf(Object o, int index): This method returns the index of the first occurrence of the specified element in this vector, searching forwards from index, or returns -1 if the element is not found.

PriorityQueue Class

1. Java PriorityQueue class is a queue data structure application in which objects are inserted based on their priority.

2. It is not like standard queues where to insert and delete the element FIFO (First-In-First-Out) algorithm is followed.

3. In a priority queue, objects are arranged according to their priority.

4. By default, the priority is determined by objects‟ natural ordering.

5. Default priority can be modified by a value called as Comparator provided during the construction time of queue.

6. If multiple objects are present of same priority the it can poll any one of them randomly.

7. It does not allow NULL objects.

SortedMap Interface

1. In Java, the SortedMap interface is part of the Java Collections Framework and extends the Map interface.

2. It maintains its elements in sorted order, typically based on the natural ordering of its keys or a custom comparator provided at the time of creation.

3. A SortedMap guarantees that the keys are maintained in ascending order according to the natural ordering of the keys, or by a custom comparator provided by the user during creation.

4. Like other map interfaces, SortedMap does not permit null keys but may permit null values. It may also provide immutability guarantees, depending on the implementation.

5. The Java standard library provides several implementations of the SortedMap interface, including TreeMap, which is commonly used. Other third-party libraries may offer additional implementations.