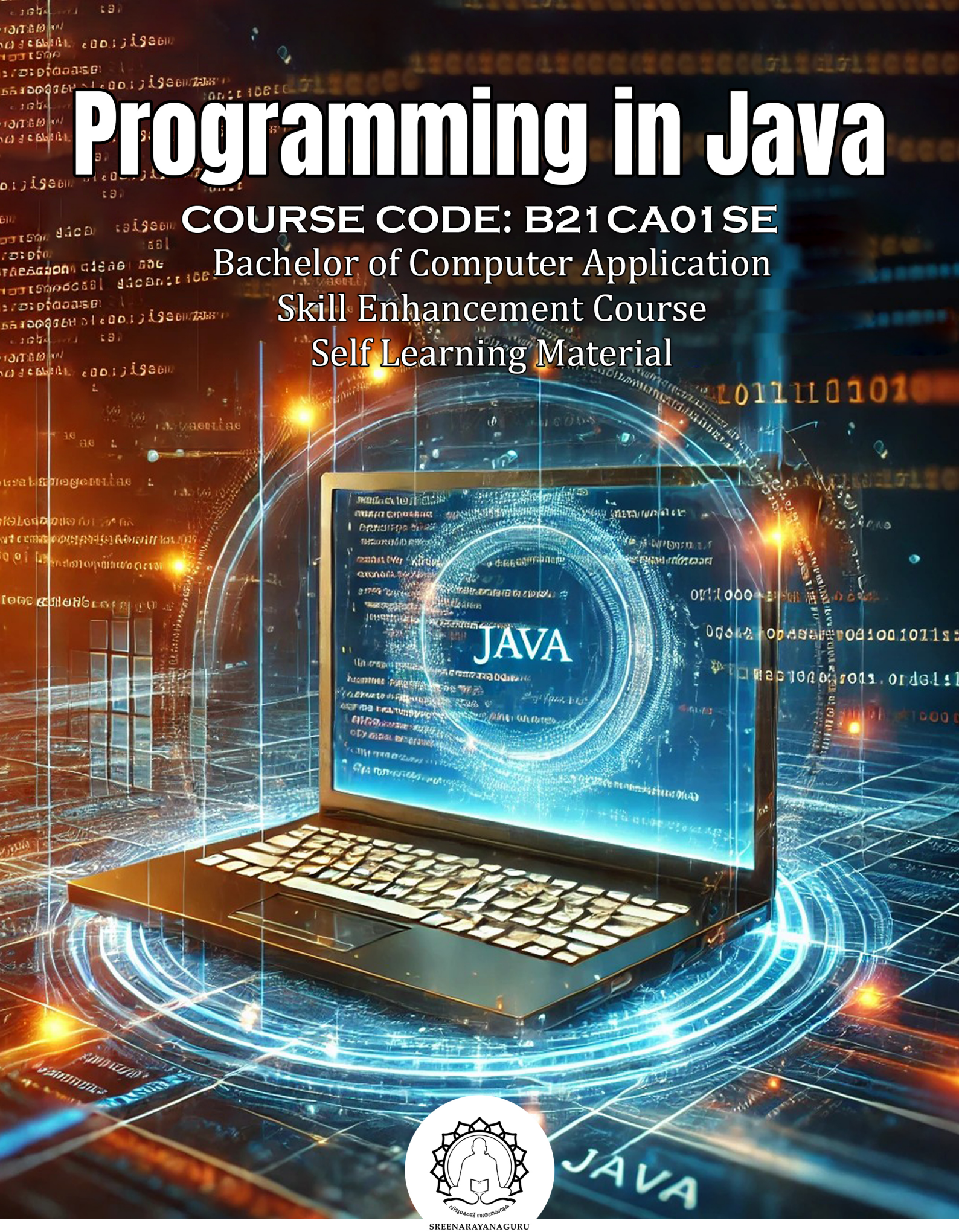
Programming in Java

COURSE CODE: B21CA01SE

Bachelor of Computer Application

Skill Enhancement Course

Self Learning Material



SREENARAYANAGURU
OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU OPEN UNIVERSITY

Vision

To increase access of potential learners of all categories to higher education, research and training, and ensure equity through delivery of high quality processes and outcomes fostering inclusive educational empowerment for social advancement.

Mission

To be benchmarked as a model for conservation and dissemination of knowledge and skill on blended and virtual mode in education, training and research for normal, continuing, and adult learners.

Pathway

Access and Quality define Equity.

Programming in Java

Course Code: B21CA01SE

Semester - II

Skill Enhancement Course
Undergraduate Programme
Bachelor of Computer Applications
Self Learning Material

SREENARAYANAGURU
OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU
OPEN UNIVERSITY

PROGRAMMING IN JAVA

Course Code: B21CA01SE
Semester- II
Skill Enhancement Course
Bachelor of Computer Applications

Development of the Content

Shamin S., Dr. Jennath H.S.,
Suramya Swamidas P.C.,
Greeshma P.P., Sreerekha V.K.,
Lekshmi A.C.

Review and Edit

Dr. Aji S.

Linguistics

Dr. Aji S.

Scrutiny

Shamin S., Dr. Jennath H.S.,
Suramya Swamidas P.C.,
Greeshma P.P., Sreerekha V.K.

Design Control

Azeem Babu T.A.

Cover Design

Jobin J.

Co-ordination

Director, MDDC :
Dr. I.G. Shibi
Asst. Director, MDDC :
Dr. Sajeevkumar G.
Coordinator, Development:
Dr. Anfal M.
Coordinator, Distribution:
Dr. Sanitha K.K.

Scan this QR Code for reading the SLM on a digital device.

Edition
January 2025

ISBN 978-81-984969-2-8

www.sgou.ac.m

Visit and Subscribe our Social Media Platforms

MESSAGE FROM VICE CHANCELLOR

Dear learner,

I extend my heartfelt greetings and profound enthusiasm as I warmly welcome you to Sreenarayanaguru Open University. Established in September 2020 as a state-led endeavour to promote higher education through open and distance learning modes, our institution was shaped by the guiding principle that access and quality are the cornerstones of equity. We have firmly resolved to uphold the highest standards of education, setting the benchmark and charting the course.

The courses offered by the Sreenarayanaguru Open University aim to strike a quality balance, ensuring students are equipped for both personal growth and professional excellence. The University embraces the widely acclaimed "blended format," a practical framework that harmoniously integrates Self-Learning Materials, Classroom Counseling, and Virtual modes, fostering a dynamic and enriching experience for both learners and instructors.

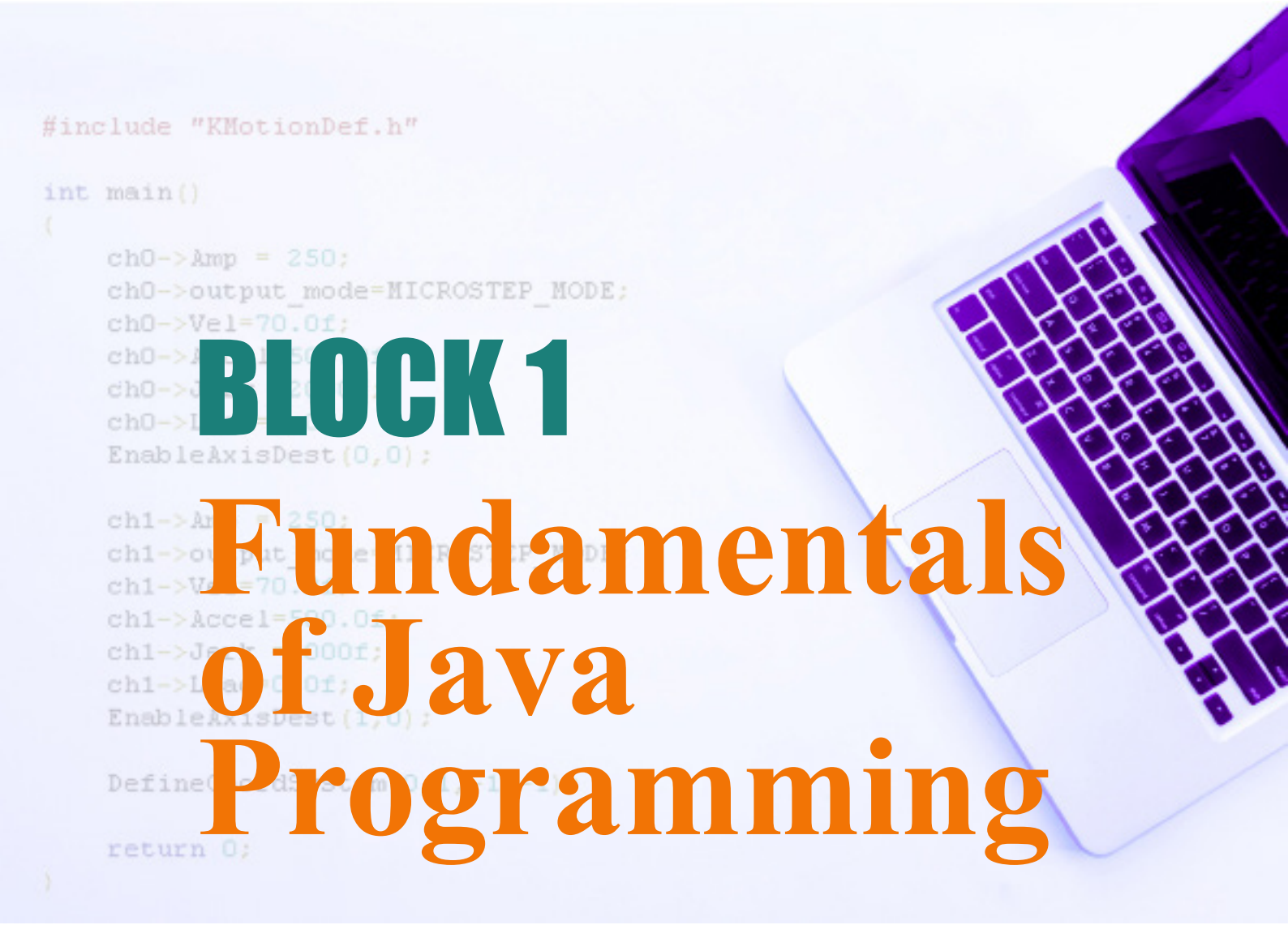
The University aims to offer you an engaging and thought-provoking educational journey. The undergraduate programme includes Skill Enhancement Courses to introduce learners to specific skills or areas related to their field of study. This is an important part of the university's plan to give learners new experiences with relevant subject content. The Skill Enhancement Courses have been designed to match those offered by other premier institutions that provide skill training. The Self-Learning Material has been meticulously crafted, incorporating relevant examples to facilitate better comprehension.

Rest assured, the university's student support services will be at your disposal throughout your academic journey, readily available to address any concerns or grievances you may encounter. We encourage you to reach out to us freely regarding any matter about your academic programme. It is our sincere wish that you achieve the utmost success.

Warm regards.
Dr. Jagathy Raj V. P.

01-01-2025

Contents



BLOCK 1

Fundamentals of Java Programming

Understanding Java, Data Types and Setting up Java Environment

Learning Outcomes

The learner will be able to:

- familiarise with the fundamental concepts of Java programming
- understand the various data types in Java, such as integers, floats, characters, and boolean
- explore the use of operators in Java, including arithmetic, relational, and logical operators
- learn the process of setting up the Java programming environment
- identify the purpose and function of the Java Virtual Machine (JVM)

Prerequisites

You have already learned the building blocks of programming - basic logic, simple algorithms, and perhaps even the structure of a few other programming languages. Now, think of Java as the next level in your journey: a tool that combines these foundational ideas with the power to build complex, scalable applications.

In your previous studies, you may have learned about different data types like integers, floating-point numbers, and strings, perhaps in a language like Python or C. As you explore Java, you'll see how data types fit into a structured framework that helps Java manage memory and run efficiently. Remember, how you used operators - like adding numbers or comparing values? Java also has these operators, but it uses them with added precision and structure.

Before jumping into Java, there's some setup involved, like installing the Java Development Kit (JDK) and configuring your environment, much like setting up any other tool or device. Additionally, Java has a unique platform called the Java Virtual Machine (JVM), which allows your code to run on any system, a powerful feature that makes Java highly adaptable.

Let's embark on this journey with Java, using the knowledge you already have and discovering the nuances that make Java a favorite in the tech world.

Keywords

Java, Object Oriented Programming, Java Ecosystem, Integer, String, Floating point, Boolean, Arithmetic operator, Relational operator, Abstraction, Inheritance, Overriding, Overloading

Discussion

1.1.1 Introduction to Java Language

Java is a high-level, object-oriented programming language that was developed by Sun Microsystems in 1995, and later acquired by Oracle Corporation. It was designed with the goal of providing a platform-independent environment for software development, making it one of the most versatile and widely-used programming languages in the world.

Java follows the "write once, run anywhere" (WORA) philosophy, meaning that code written in Java can be executed on any platform that supports the Java Virtual Machine (JVM), without the need for recompilation. This feature has played a significant role in Java's adoption across various operating systems like Windows, macOS, and Linux.

1.1.1.1 Key Features of Java

- Object-Oriented: This means that Java helps you organize your code in a way that represents real-world things. You can use "objects" in your code, which makes it easier to reuse parts of your program.
- Platform-Independent: Java programs can run on any computer that has a special program called the Java Virtual Machine (JVM). This means you don't need to rewrite your code for each different type of computer.
- Simple and Easy to Learn: Java is designed so that new programmers can pick it up quickly. If you already know other programming languages like C or C++, Java will be easier for you to understand.
- Secure: Java is built with security features that help protect your programs from viruses and other threats.
- Multithreading: Java can run many parts of a program at the same time, which helps make it faster and more efficient.
- Standard Library: Java comes with a big collection of pre-written code that you can use to solve common problems, so you don't have to write everything from scratch.

1.1.1.2 Java Ecosystem

Java is not just a programming language - it's also a collection of tools and libraries that help programmers build different types of applications:

- Java Standard Edition (SE): This is the core version of Java that most people start learning.
- Java Enterprise Edition (EE): This is used for building large business applications.
- Java Micro Edition (ME): This version is for smaller devices like mobile phones or other electronic gadgets.

Java is widely used to build different types of applications such as desktop software, websites, and even mobile apps for Android phones. Many programmers love using Java because it has been around for a long time, has great community support, and constantly gets updated to stay current.

1.1.1.3 Integrated Development Environment

An Integrated Development Environment (IDE) for Java is like a supercharged workspace for programmers. It's a software tool that brings together everything a developer needs to write, test, and fix Java code - all in one place. Instead of juggling multiple programs, the IDE combines tasks like writing code, running it, finding errors, and organizing files into one easy-to-use environment.

Some cool features of Java IDEs include:

- Code Editor: A smart text editor that highlights the syntax of your code and even suggests what comes next, making coding faster and easier.
- Compiler: This tool turns your Java code into something the computer can understand and run.
- Debugger: If your code isn't working, the debugger helps you find out why by letting you pause and examine it step by step.
- Project Management: Helps you keep everything organized, from your code files to the libraries your project needs.
- Build Automation: Works with tools like Maven or Gradle to automatically compile and organize your code, saving you time.

Examples of Java IDEs:

- Eclipse: A popular, free IDE that's great for Java development and has lots of extra features you can add on.
- IntelliJ IDEA: Known for being smart - offering helpful suggestions and detecting problems in your code before you even run it. It has both a free and a paid version.
- NetBeans: Another free option that's perfect for Java developers, especially if you're working on web apps or JavaFX.
- JDeveloper: A tool from Oracle designed to make building big Java apps easy, especially for web-based projects.

- BlueJ: A simple, easy-to-learn IDE, perfect for beginners who are just getting started with Java.

1.1.1.4 Writing Your First Java Program

Writing your first Java program is an important step toward learning to code. Java programs are created by writing instructions in a text file, which the computer will follow. Here's how to write and run a simple Java program step by step.

Step 1: Set Up Your Computer

Before you can write your Java program, you need to install something called the Java Development Kit (JDK). The JDK gives you the tools to write and run Java programs. You can also download a program called an IDE (like Eclipse or IntelliJ IDEA), which makes writing code easier.

Step 2: Write Your First Java Program

Let's start with a simple program called Hello World. This program will display the words "Hello, World!" on the screen. Here's the code you will write:

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

What This Code Does:

Class Declaration:

The program starts with the word class, followed by HelloWorld. A class is just a way to organize your code.

Main Method:

The main method is where the program starts running. The program will look for this part of the code to know what to do.

Printing:

The statement System.out.println("Hello, World!"); tells the computer to show the text "Hello, World!" on the screen.

Step 3: Compile the Program

To run your program, you first need to compile it. This means changing the code you wrote into something the computer can understand. Here's how you do it:

1. Save the file as HelloWorld.java.
2. Open the command prompt (on Windows) or terminal (on macOS or Linux).
3. Go to the folder where you saved your file.
4. Type this command to compile it:

javac HelloWorld.java

If there are no mistakes, a new file called HelloWorld.class will be created.

Step 4: Run the Program

Now that your program is compiled, you can run it. In the same command prompt or terminal, type:

java HelloWorld

You should see:

Hello, World!

This means your program worked!

How Java Programs Work:

- You write the program in a .java file (source code).
- The javac command compiles it into a .class file (bytecode).
- The java command runs the program and shows the result.

1.1.2 Java Program Structure

When you write a Java program, you need to follow a specific structure so the computer can understand and run it. This structure is the same for all Java programs, no matter how simple or complex. Let's break it down step by step.

1. Package Declaration (Optional)

At the very top of some Java programs, you might see something called a package declaration. This is like a folder where your program is stored. You only need this if your program belongs to a package, but it's optional.

package myFirstProgram;

2. Import Statements (Optional)

Sometimes, Java programs use code from other libraries, which are collections of useful code already written for you. To use these, you need to import them into your program. It's like telling Java, "Hey, I need some extra tools!"

Example:

import java.util.Scanner;

Here, the program is importing a tool to read input from the user.

3. Class Declaration

Every Java program must have a class. A class is like a container that holds the instructions for your program. The class name should match the file name (e.g., the class HelloWorld should be in a file called HelloWorld.java).

The class declaration looks like this:

```
public class HelloWorld
{
    // The rest of the program goes here
}
```

4. Main Method

The main method is where your program starts running. It's the first thing Java looks for when it starts your program. Every program needs this main method for it to work.

```
public static void main(String[] args)
{
    // Program instructions go here
}
```

5. Statements

Inside the main method, you write the actual instructions for what you want the program to do. These instructions are called statements, and each one ends with a semicolon (;). For example, if you want the program to display something on the screen, you write:

```
System.out.println("Hello, World!");
```

6. Comments

Comments are notes you can write in your code that Java ignores. They are helpful for explaining what your code does, making it easier for you (or someone else) to understand it later. There are two types of comments:

- Single-line comment: Starts with // and only comments on one line.

  Example:

  ```
  // This is a single-line comment
  ```

- Multi-line comment: Starts with /* and ends with */, so you can write comments across multiple lines.

  Example:

```
/* This is a comment across
multiple lines */
```

7. Curly Braces

Java uses curly braces {} to group parts of the program together. The opening { starts a block of code, and the closing } ends it. For example, every class and method must be inside a pair of curly braces.

Example:

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

In this program, the braces group the code into blocks for the class and the main method.

8. Optional: Other Methods and Variables

Apart from the main method, a Java program can have other methods and variables (data storage). These extra methods help organize the code and perform different tasks.

Example:

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

In this example, printMessage() is a separate method used to print the message.

1.1.3 Role of the main( ) Method

The main( ) method is a crucial part of any Java program. It's the starting point where Java programs begin to run. If you want your program to do anything, it needs a main() method to get things going.

The main() method is where Java begins to execute the code. When you run your program, Java looks for this method to know what to do first. If your program doesn't have a main() method, it won't work.

Each part of the main() method is important:

- public: This means the method can be accessed from anywhere. Java needs this so it can start running your program.
- static: This lets the main() method run without creating an object of the class. It means the method is part of the class itself, not an instance of it.
- void: This tells Java that the main() method doesn't return anything. It just runs the program.
- String[] args: This is a way for your program to accept inputs from the user when they run the program. These inputs are stored in an array of strings (text).

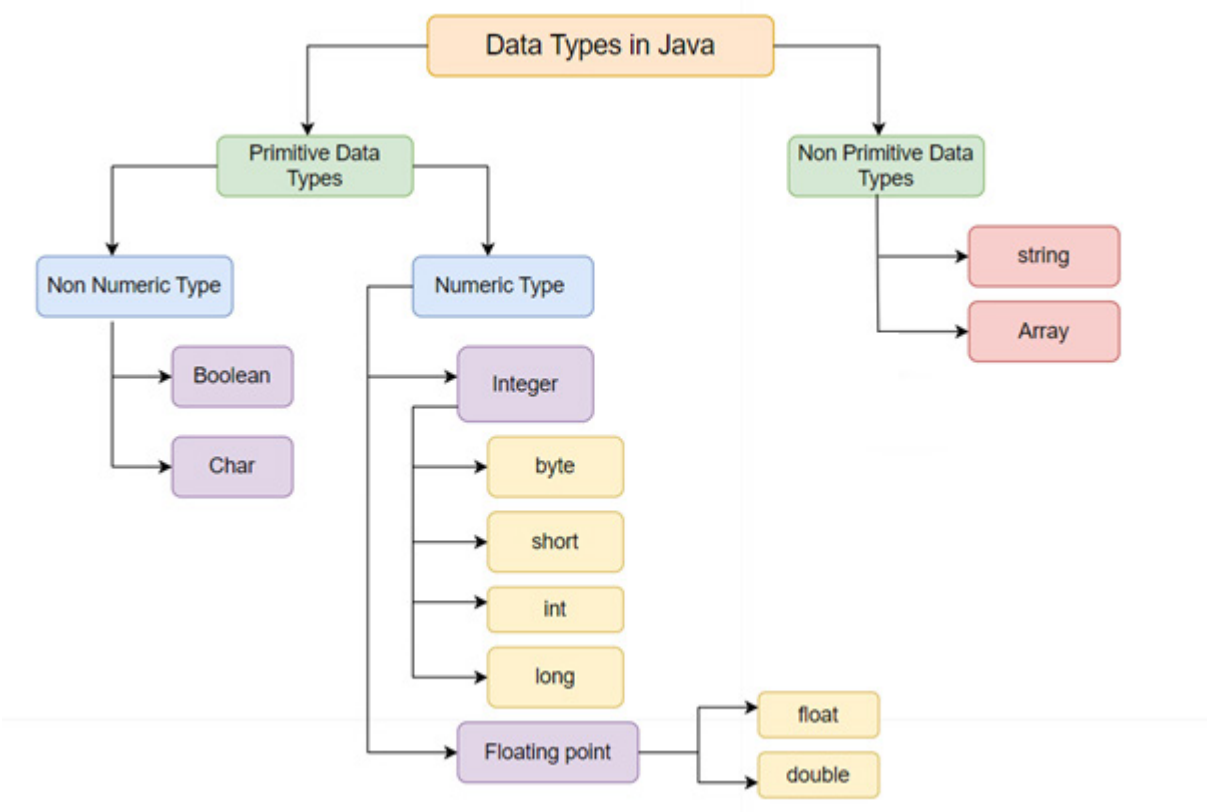


Fig 1.1.1 Classification of data types in java

The main() method is where you write the instructions for what you want your program to do. This is where the action happens! The String[] args part of the main() method allows your program to take inputs when it's started. These inputs are stored as a list of strings, and you can use them inside your program.

1.1.4 Data Types

A variable in java can store data of different size and value. The main two categories of data types in java are primitive and non primitive data types. The below given figure 1.1.1 shows different data types and their further classification.

1.1.5 Primitive data type

In Java, primitive data types serve as fundamental building blocks from which other data types and structures can be created. There are 4 main categories of primitive data types in java. The main categories of primitive data types are:

1. Integer data type
2. Floating Point data type
3. Boolean data type
4. Char data type

The integer and floating point data type are the main types of numeric data types and boolean and char data type are the main types of non numeric data types as shown in figure 1.1.1. First we will discuss Integer data types in detail.

1.1.5.1 Integer data type

Java includes four integer types: byte, short, int, and long. Each of these types can represent whole numbers, including both positive and negative values.

Table 1.1.1 Types of integer data type

Data type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372, 036,854,775, 808 to 9,223,372, 036,854, 775,807

a) byte

The byte data type is the smallest integer type, represented by 8 bits, and its value ranges from -128 to 127. Variables of byte data type are defined by using **byte** keyword. For example, the following declares two byte variables called a and b:

```
byte a, b;
```

b) short

The short data type is a signed 16-bit type, with a range spanning from -32,768 to 32,767. It is one of the least commonly used types in Java. A variable of short data type is defined by using **short** keyword. Below are some examples of how to declare short variables.

```
short a;

short b;
```

c) int

The int data type is the most frequently used integer type. It is a signed 32-bit type with a value range of -2,147,483,648 to 2,147,483,647. A variable of int data type is defined by using the keyword **int**. Below are some examples of how to declare int variables.

```
int a;

int b;
```

d) long

The long data type is a signed 64-bit type, ideal for situations where the int type cannot accommodate the required value. With its large range, long is useful for handling very large whole numbers. A variable of long data type is declared by using **long** keyword. Below are some examples of how to declare int variables.

```
long b;

long c;
```

1.1.5.2 Floating Point data type

Floating-point numbers, often referred to as real numbers, are utilized in calculations that require fractional accuracy. For example, computations like square root, logarithms or exponential functions yield values that necessitate the use of floating-point types to maintain the required level of precision. Floating-point types are mainly in two forms: float and double, representing single-precision and double-precision numbers, respectively.

a) float

The float type defines a single-precision value that occupies 32 bits of memory. It holds fractional numbers and is adequate for storing 6 to 7 digits after the decimal

point. Variables of float data type are defined by using **float** keyword. Example of float variables are shown below:

float AvgMarks;

b) double

Double precision, represented by the keyword double, utilizes 64 bits for storing a value. All transcendental mathematical functions, like sin(), cos(), and sqrt(), return values of the double type. Example of declaring a variable of double type is shown below:

double radius;

double temperature;

1.1.5.3 Char data type

In Java, the char data type is used for storing characters. The char data type in java and char data type in c/c++ language are not same. In C and C++, the char type is 8 bits in size. This is not applicable in Java. Instead, Java utilizes Unicode for character representation. The char data type is employed to store individual 16-bit Unicode characters, such as 'A', '1', or '$'. The character should be enclosed in single quotes. A variable of char data type is declared by using **char** keyword. An example of initialization of char type variable is shown below:

char letter='a';

char digit='2';

1.1.5.4 Boolean

In java the boolean data type is used to represent two possible values that is **true** or **false**. This is the type produced by all relational operators, such as in the expression a < b. The keyword boolean is used to create a variable of boolean type. An example of declaration of boolean type variable is given:

boolean a;

boolean b;

1.1.6 Non Primitive data types

A non primitive data type also known as reference type, which is used to refer to objects. Non-primitive types are defined by the programmer and are not built into Java. A Primitive data types directly contain their values within the variable, which means the variable has the actual data. On the other hand, non-primitive types do not keep the data directly; instead, they maintain a reference that indicates the memory location where the data is stored. Some examples of non primitive data types are listed below:

1. Array
2. String

1. Array

An array is a group of data values of the same type. A single array variable name can be used to store and access multiple values of the same type. The syntax for declaring an array variable is:

Data_type Array_name[array_size];

Example:

int a[10];

The above example declares an array named a that can hold 10 elements and all the data values are of integer type.

There are two types of array in java which are:

1. **Single dimensional array:** A single-dimensional array is a group of elements of the same data type arranged in a continuous segment of memory.

2. **Multi dimensional array:** A multi-dimensional array is an array that includes one or more arrays as its elements.

2. String

A string data types is a sequence of characters. The syntax for declaring string variable is:

String variable_name;

The syntax for initialization of string variable is:

String s="hello world";

1.1.7 Operators in java

An operator is used to perform specific mathematical or logical operations on values. The values that the operators work on are called operands. In java there are different types of operators.

1.1.7.1 Arithmetic Operators

The basic arithmetic operations are addition, subtraction, multiplication, and division. The following simple example program demonstrates the arithmetic operators.

Program 1

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

```
        System.out.println("Integer Arithmetic");
        int a = 1 + 1;
        int b = a * 3;
        int c = b / 4;
        int d = c - a;
        int e = -d;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
       System.out.println("c = " + c);
       System.out.println("d = " + d);
       System.out.println("e = " + e);
    }
}
```

Output

```
Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e  = 1
```

1.1.7.2 The Modulus Operator

The modulus operator, denoted as %, gives the remainder from a division operation. It can be utilized with both floating-point and integer types. The following example program demonstrates the %:

Program 2

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

```
        {
            int x = 42;
            double y = 42.25;
            System.out.println("x mod 10 = " + x % 10);
            System.out.println("y mod 10 = " + y % 10);
    }
}
```

Output

```
x mod 10 = 2
y mod 10 = 2.25
```

1.1.7.3 Arithmetic Compound Assignment Operators

Java offers unique operators that allow for the combination of an arithmetic operation and an assignment in one step. For example:

```
a = a + 2;
```

In Java, you can rewrite this statement as shown here:

```
a += 2;
```

1.1.7.4 Increment and Decrement

The ++ and the – – are Java's increment and decrement operators. The increment operator increases its operand by one. The decrement operator decreases its operand by one. For example, this statement:

```
x = x + 1;
```

can be rewritten like this by use of the increment operator:

```
x++;
```

Similarly,

```
x = x - 1;
```

is equivalent to `x--;`

1.1.7.5 Bitwise Operators

Java includes a variety of bitwise operators that can be used with integer types such as

long, int, short, char, and byte. These operators operate on the individual bits of their operands. They are summarized in the following table 1.1.2

Table 1.1.2 Bitwise Operator

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

1.1.7.6 Bitwise Logical Operators

The bitwise logical operators consist of &, |, ^, and ~. The table below illustrates the results of each operation.

Table 1.1.3 Bitwise Logical Operators

A	B	A|B	A&B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

1.1.7.7 Relational Operators

Relational operators establish the relationship between two operands, specifically assessing equality and order. The following table 1.1.4 shows the relational operators:

Table 1.1.4 Relational Operators

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The outcome of these operations is a boolean value.

1.1.7.8 Boolean Logical Operators

The Boolean logical operators displayed here function exclusively on boolean operands. All binary logical operators combine two boolean values to produce a resulting boolean value.

Table 1.1.5 Boolean Logical Operators

Operator	Result
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment

==	Equal to
!=	Not equal to
?:	Ternary if-then-else
&	Logical AND

1.1.7.9 Assignment Operator

The assignment operator is represented by a single equal sign, =. In Java, the assignment operator functions similarly to how it does in other programming languages. Its general format is as follows:

```
var = expression;
```

In this case, the type of var must be compatible with the type of expression. For example:

```
int x, y, z;
x = y = z = 100;    // set x, y, and z to 100
```

1.1.8 Java Environment Set Up

Java is a versatile programming language designed for general-purpose use. It supports concurrency, is class-based, and follows an object-oriented approach. Java applications are typically compiled into bytecode, which can be executed on any Java Virtual Machine (JVM), regardless of the underlying hardware architecture. Setting up an efficient Java development environment is crucial for streamlining the coding process and boosting productivity.

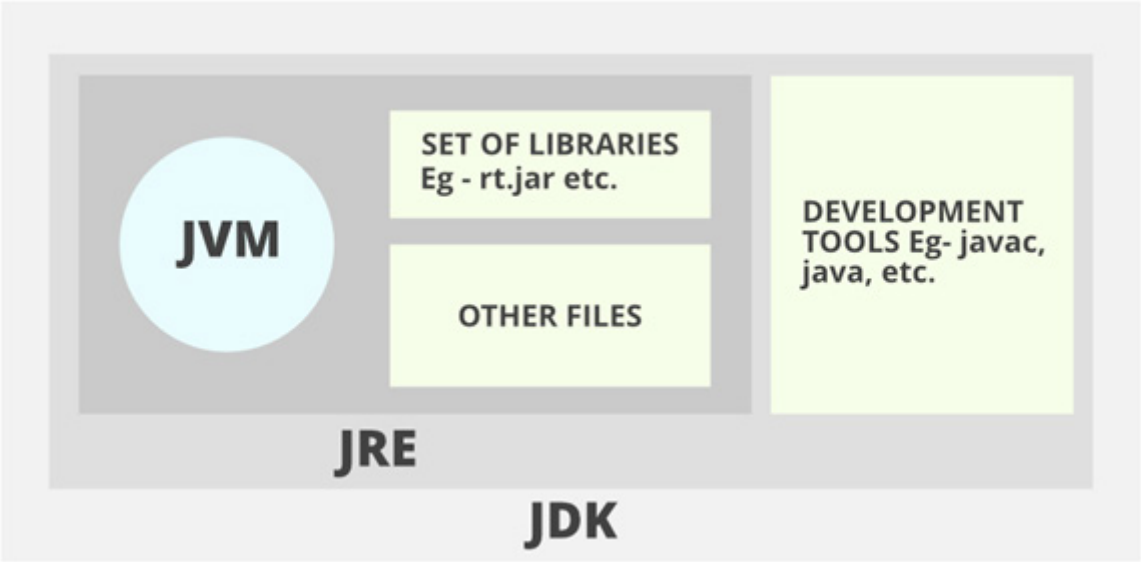


Fig 1.1.2 Java Environment

In this guide, we explain the essential steps to set up your Java environment, including installing the Java Development Kit (JDK), selecting an Integrated Development Environment (IDE), and configuring the necessary tools and libraries to ensure a smooth workflow. While the JVM, JRE, and JDK are platform-dependent due to differences in operating system configurations, Java itself remains platform-independent. Before setting up the environment, it is important to understand the JVM, JDK, and JRE. Figure 1.1.2 illustrates the relationship between the JVM, JDK, and JRE.

1.1.8.1 The Java Development Kit (JDK)

Java Development Kit (JDK) is a software development environment used for building applications, applets, and components using the Java programming language. It includes tools and libraries necessary for developing Java applications. Key Components of JDK are Java Compiler (javac), Java Runtime Environment (JRE), Java Virtual Machine (JVM), Development Tools, Javadoc, Jar, and a debugger.

JDK Versions:

Java is continuously updated with new versions. The latest version as of now is **Java 22**, which includes several new features and performance improvements.

1.1.8.2 The Java Runtime Environment (JRE)

It is a software package that provides the libraries, Java Virtual Machine (JVM), and other components needed to run Java applications and is intended for end users. It is the implementation of the runtime portion of the Java Development Kit (JDK), allowing users to run, but not develop, Java applications. JRE can be viewed as a subset of JDK. Key Components of JRE, Java Virtual Machine (JVM), Class Libraries, Class Loader, Runtime Libraries.

1.1.8.3 JVM: JVM (Java Virtual Machine)

The **Java Virtual Machine (JVM)** is a core component of the Java programming language. It is an abstract machine and also a part of the Java Runtime Environment (JRE) that executes Java programs by converting the platform-independent **bytecode** (compiled from Java source code) into machine-specific instructions. JVMs are available for many hardware and software platforms. JVM Components are, Class Loader, Runtime Data Area, Execution Engine.

Key Features of the JVM:

1. Platform Independence:

Java achieves platform independence through its compilation process. When Java source code is compiled, it is converted into bytecode, which is not tied to any specific platform. The Java Virtual Machine (JVM) interprets this bytecode and translates it into platform-specific machine code at runtime. This enables Java programs to run seamlessly on any operating system that has a compatible JVM, embodying the "Write Once, Run Anywhere" principle.

2. Memory Management:

The Java Virtual Machine (JVM) handles memory management by allocating memory, managing its usage, and performing garbage collection. This ensures that system resources are used efficiently, freeing up memory that is no longer needed and preventing memory leaks.

3. Execution Engine:

The Execution Engine is the heart of the JVM, responsible for running bytecode. It does this either by interpreting the bytecode or using Just-In-Time (JIT) compilation, which converts frequently used bytecode into native machine code to improve performance.

4. Security:

The JVM ensures a secure execution environment by implementing access controls and sandboxing, which protect against the execution of malicious code.

1.1.8.4 Java Compiler (javac):

Converts human-readable Java code (.java files) into bytecode (.class files) that the JVM can execute.

1.1.9 Setting up Java Environment

To develop and run Java programs, it is essential to set up a proper Java environment on your system. This process involves installing the necessary tools and configuring your system for a smooth development experience. .To set up a Java development environment on your machine, follow these steps:

1.1.9.1 Install JDK (Windows / macOS / Linux)

1. Go to the Oracle JDK download page or OpenJDK for an open-source version.
2. Download the appropriate JDK version for your OS.
3. Install the JDK by following the on-screen instructions.

1.1.9.2 Install on Your Operating System:

Windows:

1. Download the .exe installer file for Windows.
2. Run the installer and follow the installation wizard steps.

- During installation, ensure that the "Set JAVA_HOME" option is checked. This will set up the environment variables.

Linux :

Download the .dmg installer or use a package manager like Homebrew:

To install via Homebrew, open the Terminal and run:

brew install openjdk

Linux (Ubuntu/Debian):

Open a terminal and update the package index:

sudo apt update

1. Install OpenJDK:

sudo apt install openjdk-17-jdk

1.1.9.3 Verify Installation:

Open Command Prompt (Windows) or Terminal (macOS/Linux).

Type:

java -version and javac -vrsion.

If installed correctly, it will display the JDK version.

1.1.9.4 Set Up Environment Variables (Windows):

After installation, you need to set up the JAVA_HOME variable. The first step you need to take before setting the JAVA_HOME environment variable, is to know the installation directory of the JDK. Take note of the path where the JDK is installed on your machine.

On Windows:

1. Open Control Panel → System → Advanced System Settings.

2. Go to the Advanced tab and click on Environment Variables.

3. Under System Variables, click New and set:

- Variable Name: JAVA_HOME
- Variable Value: Path to your JDK installation (e.g., C:\Program Files\Java\ jdk-17)

4. In the Path variable under System Variables, add a new entry: %JAVA_HOME%\bin.

On macOS/Linux:

- Edit the .bashrc or .zshrc file to include

  export JAVA_HOME = /path/to/jdk

  export PATH = $JAVA_HOME/bin:$PATH

- Save the file and run source .bashrc or source .zshrc.

1.1.10 Choose an Integrated Development Environment (IDE):

We can also use an Integrated Development Environment (IDE) to simplify the process of coding in Java. An IDE is a powerful software application that combines various tools and features in a single interface, making development faster and more efficient. It provides functionalities such as code completion, which suggests and auto-completes code as you type, saving time and reducing errors.

Additionally, IDEs come with built-in debugging tools that help identify and fix issues in your code by allowing you to inspect variables, step through your program line by line, and analyze error messages in detail. They also include project management tools that organize your files, manage dependencies, and streamline workflows, especially for larger and more complex projects.

Popular IDEs like IntelliJ IDEA, Eclipse, and NetBeans offer extensive customization options and support for plugins, enabling developers to tailor the environment to their specific needs. With these features, an IDE not only enhances productivity but also makes it easier for both beginners and experienced developers to write clean and efficient Java code

- Eclipse:

Eclipse is a free, open-source IDE that is widely used for Java development. It offers extensive plugin support, making it highly customizable for various types of projects. Eclipse is ideal for large-scale enterprise applications and supports other languages like C++ and Python with additional plugins.

- IntelliJ IDEA:

IntelliJ IDEA is a feature-rich IDE known for its intelligent code assistance, streamlined interface, and productivity-boosting features. Its **Community Edition** is free and perfect for beginners, while the paid **Ultimate Edition** includes advanced tools for web and enterprise development. IntelliJ is praised for its smooth navigation, smart suggestions, and ease of use.

- NetBeans:

NetBeans is a lightweight and user-friendly IDE, making it a great choice for beginners and educational purposes. It provides an integrated environment with built-in tools for debugging, testing, and managing projects. NetBeans is also flexible, supporting multiple programming languages beyond Java, such as PHP and JavaScript.

1.1.11 Hello World Example:

To confirm your Java environment is set up correctly, write a simple Java program:

1. Open your IDE or any text editor and write a simple "Hello, World!" program

```
public class HelloWorld
{
```

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

2. Save the file as HelloWorld.java.

3. Compile it using the command:

javac HelloWorld.java

3. Run the program with:

    java HelloWorld

If the message "Hello, World!" is displayed, your setup is successful.

1.1.13 Java Virtual Machine (JVM) Platform

Java Virtual Machine (JVM) runs Java applications as a run-time engine. JVM is the one that calls the main method present in a Java code. JVM is a part of JRE(Java Runtime Environment). The JVM platform is a powerful and versatile computing environment that allows developers to write and execute programs in various languages that compile to JVM bytecode. It was originally designed to run **Java** applications but now supports many languages such as Kotlin, Scala, Groovy, Clojure, and others. Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and expect it to run on any other Java-enabled system without any adjustment. This is all possible because of JVM.

When we compile a .java file, .class files(contains byte-code) with the same class names present in the .java file are generated by the Java compiler. This .class file goes into various steps when we run it. These steps together describe the whole JVM.

1.1.13.1 Key Features of the JVM Platform

- **Platform Independence:** Java code is compiled into **bytecode**, a platform-neutral intermediate representation. The JVM interprets or compiles this bytecode into platform-specific machine code, allowing the same Java application to run on any system with a compatible JVM.
- **Security:** The JVM ensures secure execution by performing bytecode verification, access control, and sandboxing. This prevents malicious code from harming the system or accessing unauthorized resources.
- **Automatic Memory Management:** The JVM manages memory allocation and deallocation automatically through garbage collection, reducing the risk of memory leaks and improving application stability.

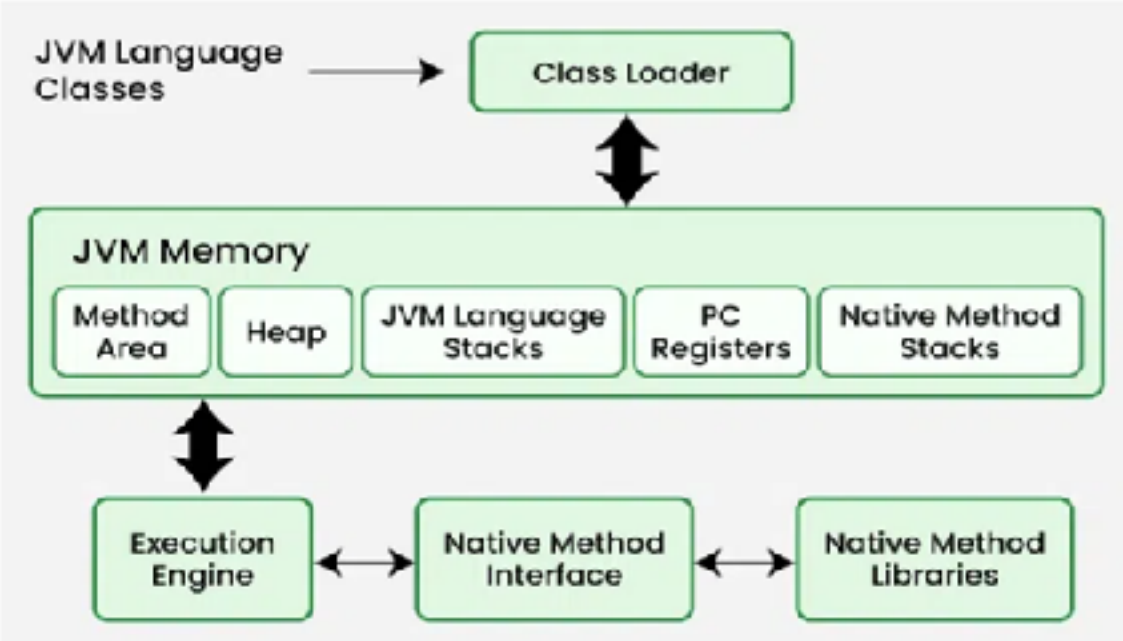


Fig 1.1.3 JVM Platform

- **Multithreading Support:** The JVM provides built-in support for multithreading, allowing Java applications to perform multiple tasks concurrently. This is crucial for modern, high-performance applications.
- **Just-In-Time (JIT) Compilation:** The JVM uses JIT compilation to optimize performance. Frequently executed bytecode is compiled into native machine code at runtime, making execution faster.
- **Interoperability with Native Code:** Through the **Java Native Interface (JNI)**, the JVM allows Java applications to interact with code written in other languages like C and C++.

Figure 1.1.3 shows the architecture of JVM Platform

Here are the key aspects of the JVM platform:

1.1.13.2 Core Components:

The core components of the JVM are designed to handle various aspects of Java application execution, such as loading, verifying, and running bytecode. Here are the key components of the JVM platform:

1.1.13.2.1 Class Loader

Class loader loads class files and bytecode into memory, enabling the JVM to dynamically load classes when required. It follows the delegation model, which ensures classes are loaded in the correct order. Key tasks include:

- Loading classes from different sources (e.g., .class files or JAR files).
- Linking classes by verifying bytecode for correctness and preparing static

variables.

- Initializing classes by executing static blocks and initializing static variables.

1.1.13.2.2 Execution Engine:

The Execution Engine is the core of the JVM, responsible for executing Java bytecode. It includes:

- **Interpreter:** Executes bytecode instructions line by line, ensuring portability but slower performance.
- **Just-In-Time (JIT) Compiler:** Compiles frequently executed bytecode into native machine code at runtime, improving performance.
- **Garbage Collector:** Automatically manages memory by reclaiming unused objects, preventing memory leaks.

1.1.13.2.3 Java Memory Area (Runtime Data Areas):

The JVM allocates memory into different areas to manage data efficiently:

- **Heap:** Stores objects and class instances; shared among all threads.
- **Stack:** Stores method call frames, including local variables and partial results; specific to each thread.
- **Method Area (or Permanent Generation in older JVMs):** Stores class-level information, such as method code and static variables.
- **Program Counter Register:** Keeps track of the current instruction address for each thread.
- **Native Method Stack:** Stores native method calls, which are written in languages like C or C++.

1.1.13.2.4 Native Method Interface (JNI):

The Java Native Interface (JNI) enables Java applications to call native methods written in languages like C or C++. This component ensures interoperability between Java and non-Java code.

1.1.13.2.5 Garbage Collection Subsystem:

The Garbage Collector manages automatic memory deallocation. It identifies objects that are no longer in use and removes them to free memory, ensuring efficient utilization of system resources.

1.1.13.2.6 Bytecode Verifier:

The Bytecode Verifier ensures the integrity and safety of the bytecode before it is executed. It checks for illegal bytecode, ensures type safety, and verifies that bytecode adheres to the Java language specifications.

1.1.13.2.7 Java Native Libraries:

The JVM uses a set of platform-specific native libraries to interact with the underlying hardware

1.1.13.3 JVM Languages:

Although Java is the primary language for the JVM, many other languages can compile down to JVM bytecode, such as:

- **Kotlin**: Modern language interoperability with Java, widely used for Android development.
- **Scala**: A functional and object-oriented programming language used for applications such as big data processing (e.g., Apache Spark).

Groovy: A dynamic language that is often used for scripting, building DSLs (domain-specific languages), and automation.

1.1.13.4 JVM in Enterprise and Cloud Environments:

JVM is extensively used in enterprise systems and cloud environments. Technologies like **Spring Framework**, **Hibernate**, and **Apache Tomcat** are commonly used for large-scale Java-based web applications and microservices. JVM is also integrated into modern cloud-native environments, with containers like **Docker** and orchestration platforms like **Kubernetes** allowing JVM-based applications to scale efficiently.

Recap

- Java is a high-level, object-oriented programming language that was developed by Sun Microsystems.
- Java follows the "write once, run anywhere" (WORA) philosophy.
- Key features of Java are Object-Oriented, Platform-Independent, Simple and Easy to Learn, Secure, Multithreading.
- Java Standard Edition (SE): This is the core version of Java that most people start learning.
- Java Enterprise Edition (EE): This is used for building large business applications.
- Java Micro Edition (ME): This version is for smaller devices like mobile phones or other electronic gadgets.
- The JDK gives you the tools to write and run Java programs.
- A class is like a container that holds the instructions for your program. In Java class name should match the file name.
- The main method is where your program starts running.

- Comments are notes you can write in your code that Java ignores.
- Java uses curly braces {} to group parts of the program together.
- Primitive Data Types in Java
- 8 different data types
- Main categories: Integer, Floating Point, Boolean, Char
- Numeric types: Integer and Floating Point
- Non-numeric types: Boolean and Char
- Operators in Java
- Arithmetic Operators
- Basic operations: +, -, *, /, and unary + and -
- Modulus Operator (%)
- Returns remainder of division
- Arithmetic Compound Assignment Operators
- Combine arithmetic operation and assignment (e.g., a += 4)
- Increment (++) and Decrement (--) Operators
- Bitwise Operators
- Include bitwise NOT, AND, OR, XOR, and shift operators
- Relational Operators
- Compare values (e.g., ==, !=, >, <, >=, <=)
- Boolean Logical Operators
- Logical OR, XOR, short-circuit AND, short-circuit OR, NOT
- Assignment Operator (=)
- Setting up java environment
- Java Development Kit (JDK)
- Java Virtual Machine (JVM)
- Java Runtime Environment (JRE)
- Installation of JDK
- JVM Platform
- Core Components of JVM

Objective Type Questions

1. Who developed Java?
2. What philosophy does Java follow?
3. What is the core version of Java?
4. What do curly braces {} do in Java?
5. What is the smallest integer data type in Java?
6. Which data type in Java is used to store decimal numbers?
7. What is the default value of a boolean variable in Java?
8. How many primitive data types are there in Java?
9. What type of data does a String variable hold?
10. Which operator is used to find the remainder of a division operation in Java?
11. Which operator is used for bitwise AND operation?
12. What type of operators compare two operands for equality?
13. What keyword is used to declare a boolean variable in Java?
14. Which operator is used for assignment in Java?
15. What is the core component responsible for loading Java classes into memory in the JVM?
16. Which of the following tools is NOT part of the JDK?
17. The JIT Compiler in the JVM is responsible for:
18. Which of the following is a feature of JVM that allows Java programs to run on any platform?
19. Which class loader is responsible for loading core Java API classes in JVM?
20. What is the correct sequence of operations during the "Linking" phase in JVM?
21. What does the acronym WORA stand for in the context of Java?
22. Which JVM component is responsible for automatic memory management?
23. In the context of Java, what is the purpose of the JAVA_HOME environment variable?
24. Which of the following is NOT a JVM language?

Answers to Objective Type Questions

1. Sun Microsystems
2. WORA
3. Java SE
4. Group code
5. byte
6. float
7. False
8. 8
9. Characters
10. %
11. &
12. Relational
13. boolean
14. =
15. Class Loader
16. Garbage Collector
17. Just-in-time compilation of bytecode to native machine code
18. Platform Independence
19. Bootstrap class loader
20. Verification → Preparation → Resolution
21. Write Once, Run Anywhere
22. Garbage Collector
23. It stores the location of the JDK installation directory
24. Python

Assignments

1. Explain the architecture of the Java Virtual Machine (JVM) in detail. Include a diagram showing the components of the JVM such as the Class Loader, Runtime Data Area, and Execution Engine.

2. Differentiate between JDK, JRE, and JVM. Provide detailed explanations of their roles, components, and how they interact with one another during the development and execution of Java programs.

3. Describe the Class Loading process in the JVM. Explain the steps involved in loading, linking, and initializing classes in the JVM. Discuss how the Bootstrap, Extension, and Application class loaders function.

4. Write a detailed step-by-step guide for setting up the Java Development Environment (JDK) on Windows and Linux systems. Include instructions for installing the JDK, setting up environment variables, and verifying the installation.

5. Discuss the significance of "Write Once, Run Anywhere" (WORA) in Java. Explain how the JVM contributes to this principle and provide examples to demonstrate Java's platform independence.

6. Explain the role of the Garbage Collector in the JVM. Discuss different types of garbage collection algorithms supported by the JVM and how garbage collection improves memory management.

7. What is the role of the Just-In-Time (JIT) Compiler in JVM? Explain how the JIT compiler works to optimize the performance of Java applications during runtime.

8. Compare and contrast the various types of JVM languages (Java, Kotlin, Scala, Groovy, etc.). Explain how these languages are compiled to JVM bytecode and discuss their interoperability with Java.

9. Illustrate how memory is managed in the JVM. Discuss the different types of memory areas such as Heap, Stack, and Method Area, and explain how they are used during the execution of a Java program.

10. Write a simple Java program that demonstrates the use of static variables and static blocks. Explain how the JVM initializes static variables and executes static blocks during class loading and initialization phases.

References

1. "Effective Java" by Joshua Bloch Edition: 3rd Edition 2018 Addison-Wesl
2. "Effective Java" by Joshua Bloch, 3rd Edition, 2018, Addison-Wesley
3. "Head First Java" by Kathy Sierra and Bert Bates, 2nd Edition, 2005, O'Reilly Media
4. "Java Concurrency in Practice" by Brian Goetz, 1st Edition, 2006 Addison-Wesley
5. "Core Java Volume I – Fundamentals" by Cay S. Horstmann, 12th Edition, 2022, Pearson

Suggested Reading

1. Herbert, Schildt. "Java: The complete Reference 9th edition." (2014).
2. Balagurusamy, Emir. Programming in Java: A Primer. McGraw-Hill Education, 2010.
3. Sierra, Kathy, and Bert Bates. Head First Java: A Brain-Friendly Guide. " O'Reilly Media, Inc.", 2005.

UNIT 2 Class, Objects and Methods

Learning Outcomes

The learner will be able to:

- familiarise the concept of Classes in java
- describe the use of objects and methods in java
- make aware of the access specifiers used in java
- narrate the uses of Static and final keywords

Prerequisites

In a car rental application, we might have a Car class that holds details like make, model, and rentalPrice. Each individual car can be represented as an object, allowing us to track individual characteristics and behaviors.

This organization simplifies code maintenance and enhances readability. For example, if we need to add new features, like a method to calculate rental costs, we can modify the Car class without affecting other parts of the program. By using classes, it promotes code reuse, as we can create multiple objects from the same class, reducing redundancy. Classes and objects help structure our code in a way that reflects real-world relationships, making it easier to develop and manage applications.

Keywords

Abstraction, Blue print, Nested class, Parameterized Constructor, Copy Constructor, Interface, Immutable

Discussion

In Java, classes and objects are central to the structure of Object-Oriented Programming (OOP), it is designed to represent real-world entities in a more understandable and practical manner. For example, a class named "Dog" would define common traits or actions that all dogs possess, such as breed, color, or methods like bark() or run(). An object, like "Tommy," would be a specific instance of the Dog class, representing one particular dog with its own unique values for the class attributes.

A class acts as a template or blueprint that defines the properties and behaviors shared by all objects of that type. Objects, on the other hand, are individual instances created from that class.

This system of defining a class and creating objects allows for efficient and scalable code, mirroring the real world by enabling the creation of multiple objects with similar characteristics but unique individual properties. By using classes and objects, Java offers a structured way to encapsulate data and behavior, making programs easier to manage, modify, and expand.

1.2.1 Understanding Java Classes

A class in Java is a foundational structure that groups together objects with similar characteristics and behaviors. It functions as a user-defined blueprint or prototype from which individual objects, or instances, are created. If we define a class named "Student," it would represent the general concept of a student, with attributes like name, age, and grade. A specific student, such as "Ravi," would be an object or instance of the "Student" class, containing unique values for these attributes. This encapsulation of data (attributes) and methods (behaviors) within a class is a key principle of Object-Oriented Programming (OOP) in Java.

1.2.1.1 Properties of java Classes

A Java class acts as a blueprint or template for creating objects, defining the shared characteristics and behaviors they will possess. It's a conceptual construct that doesn't exist physically in memory until an object is instantiated from it. Unlike objects, which occupy memory space, a class itself doesn't require any memory allocation. Its primary function is to provide a framework for future objects. Within a class, data members (variables) and methods (functions) are organized. Variables store the specific data or state of an individual object, while methods define the actions or behaviors that the object can perform. This structural organization promotes code clarity, maintainability, and reusability.

1.2.1.2 A Class in Java can Contain :

1. **Data members** are variables that store the specific characteristics or state of an object. In the "Car" class, for example, data members might include attributes like "color," "model," and "engineType." These variables define the unique properties of each car object.

2. **Methods** are functions that define the actions or behaviors that an object can perform. The "Car" class might have methods like "drive()," "brake()," and "turn()." These methods specify how a car moves, stops, or changes direction.

3. **Constructors** are special methods that are automatically called when a new object is created. They initialize the data members of the object to their default or specified values, ensuring that each instance of the class starts in a valid state.

4. **Nested classes** are classes that are defined within another class. They provide a way to logically group related components, improving code organization and encapsulation. For instance, a "Car" class might have a nested "Engine" class to represent the engine's specific properties and functions.

5. **Interfaces** are contracts that define a set of methods that a class must implement. By implementing an interface, a class can adhere to a specific behavior or functionality without having to inherit from a particular class. This promotes flexibility and reusability, allowing unrelated classes to share common behavior.

Through the use of classes, Java organizes complex systems into modular components, improving code clarity, reusability, and scalability.

1.2.1.3 Example of a Java Class

To demonstrate the concept of classes in Java, let's consider the example of a class named **"Book"**. In this case, the **Book** class will have attributes (data members) such as the title, author, and year of publication. Additionally, the class will include a constructor to initialize these properties and a method to display the book's information.

```
public class Book
{
    // Data members
    String title;
    String author;
    int yearPublished;
  // Constructor
  public Book(String title, String author, int yearPublished)
  {
        this.title = title;
        this.author = author;
```

```
        this.yearPublished = yearPublished;
    }
    // Method to display book information
    public void displayInfo()
    {
        System.out.println("Title: " + title);
        System.out.println("Author: " + author);
        System.out.println("Year Published: " + yearPublished);
    }
}
```

In this example, the Book class has three main components:

1. Data Members:

   Title: Stores the title of the book.

   Author: Stores the author's name.

   YearPublished: Holds the year when the book was published.

2. Constructor: The constructor Book(String title, String author, int yearPublished) is used to initialize a new Book object with specific values for the title, author, and publication year when it is created. This ensures that every book object starts with proper values.

3. Method: The method displayInfo() prints the book's details (title, author, year of publication) to the console, allowing us to view the book's information in a readable format.

For example, you could create an instance of the Book class like this:

```
public class Main
{
    public static void main(String[] args)
    {
        // Creating a new Book object
        Book myBook = new Book("1984", "George Orwell", 1949);
        // Displaying the book's information
```

```
        myBook.displayInfo();
    }
}
```

In this instance, the Book class encapsulates the properties and behaviors of a book. The object myBook is created from the Book class with specific values ("1984," "George Orwell," 1949), and the displayInfo() method outputs these details.

1.2.1.4 Significance of classes and objects in java

Classes and objects in Java are essential components that help developers structure their code in an organized, modular, and reusable manner. By defining classes, programmers can create blueprints for real-world entities (like books) and easily generate multiple instances (objects) that share the same structure but have unique data. This approach is a core principle of Object-Oriented Programming (OOP), allowing developers to write scalable, maintainable, and efficient code.

1.2.2 Methods in Java

In Java, a method is a collection of statements that perform a specific task and may return a result to the caller. Methods allow programmers to define reusable code that can be invoked as needed throughout a program. Unlike other programming languages like C, C++, and Python, in Java, methods must always be defined within a class. Methods in Java are comparable to functions in other languages, representing the behavior of an object and encapsulating the logic required to perform a task.

A method is essentially a set of instructions that can be executed when the method is called, providing the ability to structure code into reusable, modular components. One of the key benefits of using methods is that they prevent repetition of code, allowing for better code reusability and optimization.

Syntax of a Method definition:

```
<access_modifier><return_type><method_name>(list_of_parameters)
{
    //body of the method
}
```

The syntax of a Java method involves several components, including the access modifier, return type, method name, and optional parameters. The access modifier defines the visibility of the method (whether it can be accessed from other classes or within the same class), while the return type specifies the type of data the method will return. If the method does not return any value, the return type is set to void.

1.2.2.1 Advantages of java methods

Java methods offer several significant advantages that enhance the efficiency and

maintainability of code. One of the primary benefits is code reusability; once a method is defined, it can be called and utilized in multiple locations throughout the program, which saves developers time and effort by eliminating the need to write the same code repeatedly. This feature is especially useful in large projects, where certain operations may need to be performed multiple times. Additionally, using methods contributes to code optimization by allowing programmers to break complex tasks into smaller, more manageable segments. This not only helps avoid redundant coding but also makes it easier to read and maintain the program. Furthermore, when a method needs to be updated or fixed, developers can do so in one place without having to search through the entire codebase, thereby enhancing overall program efficiency and reducing the likelihood of errors. Overall, leveraging methods in Java promotes cleaner, more organized code, making it simpler for developers to collaborate and build robust applications.

Java methods act as time-saving constructs, enabling developers to write code that can be used again without retyping. This improves both efficiency and readability within the program.

1.2.2.2 Key Components of method declaration

In Java, a method declaration typically comprises six essential components that define how the method operates and how it can be accessed.

The first component is the **modifier**, which indicates the access level or visibility of the method. Java provides four types of access specifiers: **public** methods can be accessed from any other class, **protected** methods are accessible within their own class and subclasses, **private** methods are restricted to the defining class, and methods with **default** access (no modifier specified) can only be accessed within the same package.

Next is the **return type**, which specifies the data type of the value that the method will return. If the method does not return a value, the return type is declared as **void**. This return type is mandatory and must be clearly defined in the method syntax. Following the return type is the **method name**, which serves as the identifier used to call the method. The naming conventions for methods are similar to those for variables, but it is customary for method names to start with a lowercase letter and use camelCase for readability. The fourth component is the **parameter list**, which is a comma-separated list of inputs that the method can accept, along with their respective data types. If no parameters are required, the parentheses will remain empty. While parameters are optional, they are crucial for passing data into the method and enabling it to operate on different inputs. The other one is the **exception list**, which is optional and allows the method to declare any exceptions that it might throw during execution. This is particularly important for robust error handling, as it informs the caller about potential issues that may arise. Lastly, the **method body** is the block of code enclosed in curly braces that contains the actual logic of the method. This is where the operations intended to be performed by the method are written, defining how the method processes its inputs and what it returns. Together, these components form a complete method declaration, making it a fundamental aspect of programming in Java.

Example of a Simple Java Method:

```
public class Geometry
{
// Method to calculate the area of a rectangle
        public double calculateArea(double length, double breadth)
        {
                return length * breadth; // Returns the area of the rectangle
        }
        public static void main(String[] args)
        {
                Geometry geometry = new Geometry();
                double area = geometry.calculateArea(5.0, 10.0); // Calling the calcu
                lateArea method
                System.out.println("The area of the rectangle is: " + area);
         }
}
```

In this example, the class Geometry contains a method named calculateArea() that takes two parameters (double length and double breadth) and returns their area. The method is reusable; it can be called anytime you need to find the area of a rectangle, demonstrating both code reusability and optimization. Fig. 1.2.1 shows syntax for Java method declaration.

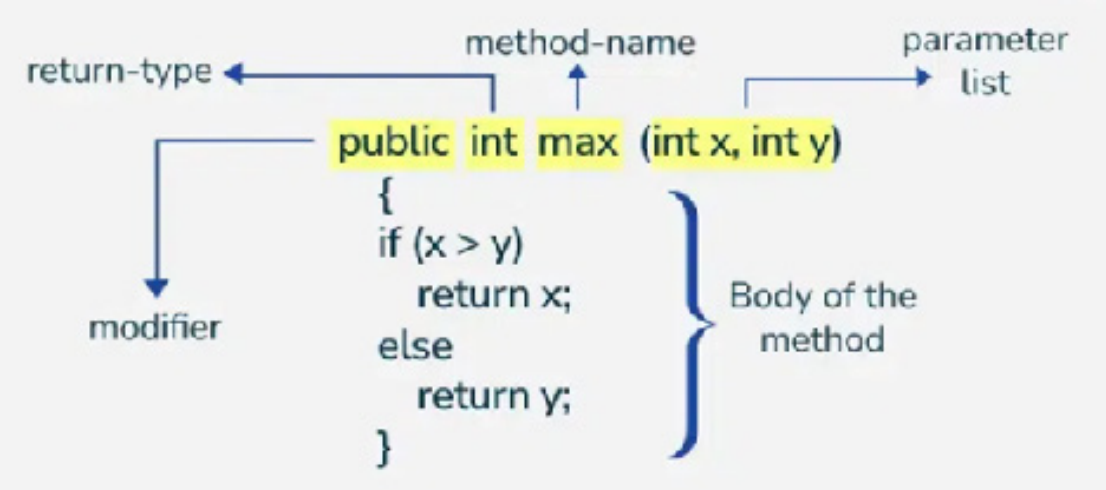


Fig. 1.2.1 Java method declaration

1.2.2.3 Types of methods in java

In Java, methods are primarily classified into two main types:

- Predefined methods
- User-defined methods

Predefined methods are those that come built into Java's class libraries, also known as the standard library or built-in methods. Java offers an extensive collection of these methods, which can be readily utilized in a program without requiring any additional code from the user. For instance, methods like System.out.println(), which is used to print output to the console, and Math.max(), which determines the maximum of two numbers, are examples of predefined methods. These methods save developers time and effort by providing ready-to-use functionality that addresses common programming tasks, making the development process more efficient.

On the other hand, user-defined methods are those crafted by the programmer to fulfill specific tasks tailored to the application's needs. These methods provide a high degree of customization and flexibility, allowing programmers to define their own operations, parameters, and return types. For example, a programmer might create a user-defined method to calculate the area of a circle, which could take the radius as a parameter and return the calculated area. This ability to design methods that address particular requirements empowers developers to create more modular and maintainable code, as they can encapsulate specific functionalities within their custom methods. Both predefined and user-defined methods play a crucial role in Java programming, with predefined methods offering convenience and efficiency, while user-defined methods provide adaptability and specificity to meet the unique demands of applications.

Ways to Create Methods in Java

Java allows methods to be created in two main ways:

1. Instance Method:

An instance method is tied to an instance of a class (i.e., an object). It can access instance data (variables that belong to the object) using the object's name. These methods are declared inside the class and can be called after an object of the class is created. Syntax:

```
void method_name()
{
        // instance area
}
```

2. Static Method:

A static method is associated with the class itself, rather than an instance of the class. It can access static data (variables that belong to the class) using the class

name. Static methods are declared inside the class using the static keyword. Syntax:

```
static void method_name()
{
        // static area
}
```

1.2.2.4 Method Signature

The method signature in Java is a unique identifier for a method, comprising the method's name and its parameter list (which includes the number of parameters, their types, and their order). The return type and exceptions thrown by the method are not part of the method signature.

Example:

For the method max(int x, int y), the signature includes two parameters of type int.

Naming a Method

When naming a method in Java, it is important to follow specific conventions to ensure readability and consistency. Method names should usually be verbs written in lowercase. If the method name contains multiple words, each subsequent word should begin with an uppercase letter (camel case), but the first word should remain in lowercase.

Rules for Naming a Method:

- The method name must be a verb and start with a lowercase letter.
- If the method name consists of more than one word, the first word can be a verb, followed by an adjective or noun.
- In multi-word method names, capitalize the first letter of each word except the first one (e.g., findSum(), computeMax(), setX(), getX()).

While methods typically have unique names within a class, Java allows multiple methods to share the same name through method overloading. This means methods with the same name can exist within the same class as long as they have different parameter lists (number, types, or order of parameters).

Example of Methods:

```
public class Calculator
{
        // Instance method
        public int add(int a, int b)
```

```
    {
        return a + b; // Adds two numbers
    }
    // Static method
    public static int multiply(int a, int b)
    {
        return a * b; // Multiplies two numbers
    }
    public static void main(String[] args)
    {
        Calculator calc = new Calculator(); // Creating an object
        int sum = calc.add(5, 10); // Calling instance method
        int product = Calculator.multiply(5, 10); // Calling static method
        System.out.println("Sum: " + sum); // Outputs: Sum: 15
        System.out.println("Product: " + product); // Outputs: Product: 50
    }
}
```

In this example both an instance method (add()) and a static method (multiply()). The instance method requires an object of the class to be called, while the static method is called directly using the class name.

By following these principles, developers can create methods that are efficient, reusable, and easy to understand, enhancing both code quality and maintainability.

1.2.2.5 Method Calling in Java

In Java, methods must be called to utilize their functionality. There are three scenarios in which a method is called:

1. The method completes all the statements within its body.
2. It reaches a return statement that sends back a value to the caller.
3. An exception occurs, which interrupts the method execution.

Let's consider an example to illustrate method calling:

```
class Addition
{
```

```
        int sum = 0;
        // Method to add two integers
        public int addTwoInt(int a, int b)
        {
                sum = a + b;
                return sum;
        }
}
public class Main
{
        public static void main(String[] args)
        {
                Addition add = new Addition();
                int result = add.addTwoInt(1, 2); // Calling the method
                System.out.println("Sum: " + result);
                // Output: Sum: 3
        }
}
```

In the above example, the addTwoInt() method is called within the main() method to add two integers. The method performs its task, returns the result, and control goes back to the calling code.

Method Calling via Different Approaches

Java supports various ways to call methods, including calling instance methods and static methods.

Example of calling a method via object creation:

```
class Test
{
 public static int count = 0;
// Constructor
        Test()
```

```
    {
        count++;
    }
    public static int getCount()
    {
        return count;
    }
    public int instanceMethod()
    {
        System.out.println("Inside instance method");
        this.staticMethod();
        return 1;
    }
    public void staticMethod()
    {
        System.out.println("Called static method");
    }
}
public class Main
{
    public static void main(String[] args)
    {
        Test obj = new Test(); // Creating an object of Test
        int result = obj.instanceMethod(); // Calling instance method
        System.out.println("Control returned after instanceMethod: " + result);
        int noOfObjects = Test.getCount(); // Calling static method
        System.out.println("Number of objects created: " + noOfObjects);
    }
}
```

In this example:

The instanceMethod() calls another method within the same class (staticMethod()).

Static methods, like getCount(), can be called directly using the class name without creating an object.

1.2.2.6 Passing Parameters to methods

Java offers several mechanisms to enhance the flexibility of method parameters:

- Passing Arrays: You can pass an entire array as an argument to a method. This allows you to pass a variable number of elements without explicitly declaring each one individually.
- Variable Arguments (Varargs): Using varargs, a method can accept a variable number of arguments of the same type. This is particularly useful when you don't know the exact number of arguments needed beforehand.
- Method Overloading: Multiple methods can have the same name but different parameter lists. This allows you to create methods with the same name but different behaviors based on the types or number of arguments passed.

Example of parameter passing in methods in Java

```
public class Example
{
        private int number;
        private String name;
        // Get methods
        public int getNumber()
        {
                return number;
         }
        public String getName()
        {
                return name;
        }
        // Set methods
        public void setNumber(int number)
```

```
    {
        this.number = number;
    }
    public void setName(String name)
    {
        this.name = name;
    }
    // Method to print details
    public void printDetails()
    {
        System.out.println("Number: " + number);
        System.out.println("Name: " + name);
    }
}
public class Main
{
    public static void main(String[] args)
    {
        Example example = new Example();
        example.setNumber(123); // Calling setter method
        example.setName("John"); // Calling setter method
        example.printDetails(); // Calling method to display details
    }
}
```

In this example, the methods setNumber(), setName(), and printDetails() are used to interact with an object of the Example class.

1.2.2.7 Advantages of using methods in java

Using methods in Java offers several significant advantages that enhance the overall quality of code and the efficiency of the development process.

Reusability is one of the primary benefits of methods; once a method is written, it

can be called multiple times throughout the program, reducing code duplication and saving development time. This not only streamlines the coding process but also ensures consistency in how certain tasks are performed, minimizing the likelihood of errors.

Abstraction is another key advantage, as methods can encapsulate complex logic, allowing programmers to use descriptive method names instead of diving into intricate code details. This abstraction makes it easier for developers to understand the code at a glance, which is particularly helpful when collaborating on larger projects with multiple contributors.

Breaking code into smaller, well-named methods enhances **readability**. Clear method names serve as documentation, indicating the purpose and functionality of each method, which helps others (or even the original developer at a later date) quickly grasp what the code is intended to accomplish.

Methods promote **encapsulation** by containing specific logic within defined boundaries, making it simpler to manage changes. When a modification is necessary, developers can update the method without affecting other parts of the program, thus enhancing maintainability.

Methods also facilitate a **separation of concerns**, allowing developers to assign different tasks to different methods. This organized approach improves the overall structure of the code, making it easier to navigate and debug.

Methods contribute to **modularity** by breaking down larger problems into smaller, manageable units. Each method can address a specific aspect of a problem, making it easier to develop, test, and maintain individual components of a larger application.

Improved testability is another significant advantage, as isolating functionality within methods allows for focused testing of individual components. Developers can create unit tests for each method, ensuring that each piece of functionality works as intended without the need to run the entire program.

It is a well-organized method that can enhance **performance** by optimizing execution time and improving code management. By reducing redundancy and focusing on efficient code structure, methods can lead to better-performing applications that are easier to scale and maintain.

1.2.3 Constructors

In Java, constructors are special methods used to initialize objects. When an object is created, the constructor is automatically called. Constructors can be used to set initial values for object attributes.

Example:

```
public class Car
{
        String model;
```

```
    int year;
    // Constructor
    public Car(String model, int year)
    {
        this.model = model;
        this.year = year;
    }
    // Display car details
    public void display()
    {
        System.out.println("Model: " + model + ", Year: " + year);
    }
}
public class Main
{
    public static void main(String[] args)
    {
        Car car1 = new Car("Toyota", 2021); // Constructor called
        car1.display();
        // Output: Model: Toyota, Year: 2021
    }
}
```

In this case, the constructor initializes the Car object's attributes when it's created, demonstrating the importance of constructors in setting up object states.

A constructor is named as such because it initializes values when an object is created. Writing a constructor for a class is optional. If no constructor is defined in the class, a default constructor is automatically called. In this case, the Java compiler generates a default constructor by default.

1.2.3.1 Need for Constructors in java

Consider the Box class. If we define a Box class, it will have variables such as length, breadth, and height. When we create an object of this class (i.e., when the Box exists

in the computer's memory), can the box have undefined dimensions? The answer is no. Constructors are necessary to assign values to these class variables when the object is created. This can either be done explicitly by the programmer or automatically by Java through a default constructor.

Every time an object is created using the new() keyword, at least one constructor (which could be the default constructor) is called to initialize the class's data members.

There are specific rules for writing constructors:

- The constructor must have the same name as the class it belongs to.
- In Java, a constructor cannot be abstract, final, static, or synchronized.
- Access modifiers can be applied to a constructor to control which classes can access it. Constructors, like methods, consist of a set of statements that execute when the object is created, helping to set the object's initial state.

1.2.3.2 Difference Between constructors and methods in java

- The constructor is used to initialize the state of an object, while a method is used to expose the behavior of an object.
- Constructors must have the same name as the class they are defined in, while methods in Java can have any name.
- Constructor is invoked implicitly, while method is invoked explicitly.
- Constructors do not have a return type, whereas methods must specify a return type or use void if they do not return a value.
- Constructors are invoked only once when an object is created, whereas methods can be called multiple times throughout the program.

Example of Java Constructor

```
// Java Program for Constructor

import java.io.*;

// Driver Class

class Box

{

        // Constructor

        Box()

        {

                super();
```

```
            System.out.println("Constructor is Called");
        }
        // main function
        public static void main(String[] args)
        {
            Box box1 = new Box();
        }
}
```

Output

Constructor is Called

The first line of a constructor is either a call to super() or this(), which invokes a constructor from the superclass or an overloaded constructor. If you don't explicitly include a call to super, the compiler automatically inserts a no-argument call to super as the first line. The superclass constructor must always be invoked to create an object. Even if your class doesn't explicitly extend another class, it is still a subclass of the Object class in Java, as all classes inherit from Object by default.

1.2.3.3 Types of constructors in java

There are three main types of constructors in Java: Default Constructor, Parameterized Constructor, and Copy Constructor.

Default Constructor in Java:

A default constructor is a constructor that has no parameters. It is automatically provided by the compiler if no constructor is explicitly written in the class. However, if we write a constructor with no arguments, the compiler does not generate a default constructor; instead, it is considered as overloaded, turning it into a parameterized constructor. This means a default constructor can be either implicit (automatically generated) or explicit (manually written). If a parameterized constructor is defined, the default constructor is no longer available unless explicitly declared. Fig 1.2.2 shows default constructor.

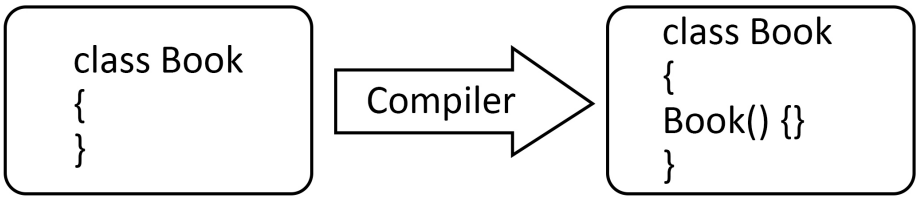


Fig 1.2.2 Default constructor

```
// Example program for Default Constructor
import java.io.*;
// Driver class
class Area
{
        // Default Constructor
        Area() { System.out.println("Default constructor"); }
        // Driver function
        public static void main(String[] args)
        {
                Area area1 = new Area();
        }
}
```

Output :

Default constructor

Parameterized Constructor in Java:

A constructor that accepts parameters is called a parameterized constructor. It is used when you want to initialize the class fields with specific values provided at the time of object creation. By passing arguments to the constructor, you can customize how the object is initialized, ensuring that the class variables have user-defined values.

```
// Java Program for Parameterized Constructor
import java.io.*;
public class Area
{
        // Data members to store length and width
        private double length;
        private double width;
        // Parameterized constructor to initialize length and width
        public Area(double length, double width)
        {
```

```
        this.length = length; this.width = width;
        System.out.println("Area object created for a rectangle with length: " +
        length + " and width: " + width);
    }
    // Method to calculate the area of the rectangle
    public double calculateArea()
    {
        return length * width;
    }
    public static void main(String[] args)
    {
        Area rectangle = new Area(5.0, 10.0); // Create an Area object for a
        rectangle with length 5.0 and width 10.0
        double area = rectangle.calculateArea();
        System.out.println("The area of the rectangle is: " + area);
    }
}
```

Output

Area object created for a rectangle with length: 5.0 and width: 10.0

The area of the rectangle is: 50.0

Copy Constructor in Java:

A copy constructor is unique in that it takes another object as a parameter and copies the data from that object into the newly created one. Unlike languages like C++ that provide built-in copy constructors, Java does not have an inbuilt copy constructor. It can be manually created, a copy constructor by passing an object of the same class and copying its values into the new object. This allows for creating duplicates of objects with the same state.

Example of Copy Constructor

```
public class Area
{
    // Data members to store length and width
    private double length;
```

```
private double width;

// Parameterized constructor to initialize length and width

public Area(double length, double width)

{

        this.length = length;

        this.width = width;

        System.out.println("Area object created with length: " + length + " and
        width: " + width);

}

// Copy constructor to create a new object with the same values as an existing
object

public Area(Area other)

{

        this.length = other.length;

        this.width = other.width;

        System.out.println("Copy constructor called. Creating a new Area
        object  with the same values.");

}

// Method to calculate the area of the rectangle

public double calculateArea()

{

        return length * width;

}

public static void main(String[] args)

{

        Area rectangle1 = new Area(5.0, 10.0);

        // Create an Area object

        Area rectangle2 = new Area(rectangle1);

        // Create a copy of rectangle1 using the copy constructor

        System.out.println("Rectangle1 area: " + rectangle1.calculateArea());
```

```
        System.out.println("Rectangle 2 area: " + rectangle2.calculateArea());
    }
}
```

Output:

Area object created with length: 5.0 and width: 10.0

Copy constructor called. Creating a new Area object with the same values.

Rectangle 1 area: 50.0

Rectangle 2 area: 50.0

The program working steps:

1. Class Definition: The Area class is defined, containing data members for length and width, a parameterized constructor, a copy constructor, and a method to calculate the area.
2. Object Creation:

- An Area object named rectangle1 is created using the parameterized constructor with length 5.0 and width 10.0.
- A second Area object named rectangle2 is created using the copy constructor, passing rectangle1 as an argument.

3. Copy Constructor Execution:

- The copy constructor is called, copying the length and width values from rectangle1 to rectangle2.

4. Area Calculation:

- The calculateArea method is called on both rectangle1 and rectangle2 to calculate their areas.

5. Output:

- The areas of both rectangles are printed. Since rectangle2 is a copy of rectangle1, both have the same length and width, resulting in the same area.

The program demonstrates how the copy constructor can be used to create a new object that is a deep copy of an existing object.

1.2.4 Access Specifier / Modifiers in java

In Java, access modifiers control the visibility and accessibility of classes, constructors, variables, methods, and data members. They play a key role in ensuring security and restricting access depending on the modifier applied. Understanding and using access modifiers correctly is essential for defining the scope of different elements in a program.

There are four main types of access modifiers in Java:

1. **Default Access Modifier**
2. **Private**
3. **Protected**
4. **Public**

1.2.4.1 Default Access Modifier:

When no specific access modifier is mentioned for a class, method, or data member, it defaults to the "default" access modifier. This means that such elements are accessible only within the same package. For example, a class in one package cannot be accessed by another class in a different package if it has default access.

Example Program for default access modifier

```
// Java program to illustrate default modifier
package p1;
// Class Greet is having Default access modifier
class Greet
{
        void display()
        {
                System.out.println("Hello World!");
        }
}
// Java program to use a class from different package with default modifier
package p2;
import p1.*;
// This class is having default access modifier
class GreetNew
{
        public static void main(String args[])
        {
                // Accessing class Greet from package p1
                Greet obj1 = new Greet();
                obj1.display();
```

```
    }
}
```

Output:

Compile time error

1.2.4.2 Private Access Modifier

The private access modifier is used by adding the keyword private. When methods or data members are declared as private, they are only accessible within the class where they are defined. Other classes, even if they belong to the same package, cannot access these private members. Top-level classes or interfaces cannot be declared as private because private restricts visibility to within the same class only. This modifier is mainly used for class-level encapsulation and cannot be applied to top-level classes, only to nested classes. For example, if two classes A and B are in the same package, and class A has a private method, class B will not be able to access that method.

```
// Java program to use class from different package with a private access specifier

package p1;

class First

{

    private void display()

    {

        System.out.println("Hello from class First ");

    }

}

class Second

{

    public static void main(String args[])

    {

        First firstobj = new First();

        // Trying to access private method of class First

        firstobj.display();

    }
```

```
}
```

Output:

error: display() has private access in First firstobj.display();

1.2.4.3 Protected Access Modifier

The protected access modifier in Java is defined using the keyword protected.

Java **packages** group related classes and interfaces together to keep code organized. They also help avoid name conflicts and control access to code through different access modifiers.

Members declared as protected can be accessed within the same package and by subclasses in other packages. This provides a level of accessibility beyond the private modifier, allowing controlled sharing of data among related classes through inheritance, while still restricting access from unrelated classes outside the package.

For example, if we have two packages, p1 and p2, and a class A in p1, we can make class A public to allow access from other packages. Suppose class A has a protected method display(). If class B in p2 extends class A, the display() method can be accessed by class B through inheritance, even though they are in different packages. The protected access enables the sharing of class functionality across package boundaries through inheritance while still maintaining encapsulation for non-subclass external classes. By creating an object of class B, the protected method display() in class A can be accessed and executed.

```
// Java Program to Illustrate Protected access specifier

package p1;

public class First
{
        protected void display()
        {
                System.out.println("Hello from class First");
        }
}

package p2;

// importing all classes in package p1

import p1.*;

// Class Second is subclass of class First

class Second extends First
```

```
{
        public static void main(String args[])
        {
                Second secondobj = new Second();
                secondobj.display();
        }
}
```

Output:

Hello from class First

1.2.4.4 Public Access Modifier

The public access modifier in Java is defined using the keyword public, and it offers the broadest scope of all access modifiers. When a class, method, or data member is declared as public, it becomes accessible from any part of the program, including across different packages. Unlike other access modifiers, there are no restrictions on public members, meaning that any class, regardless of its location, can access these public elements.

This level of accessibility is useful for methods and classes that are intended to be universally available throughout an application. The core utility classes in Java like System or Math are declared as public, allowing developers to use them in any part of their codebase without concern for package boundaries. Similarly, if a developer creates a public method in one class, any other class, whether it's in the same package or a completely different package, can invoke that method without needing special permissions.

Public access is typically used for components that are designed to provide services or functionalities that should be available throughout an application, ensuring ease of access and integration across various parts of a program. However, using public access extensively should be done with caution, as it reduces encapsulation, making code more exposed and harder to control or modify.

Program 1:

```
// Java program using public modifier

package p1;

public class First
{
        public void display()
        {
```

```
                System.out.println("Hello from class First");
        }
}
```

Program 2:

```
package p2;
import p1.*;
class Second
{
        public static void main(String args[])
        {
                First firstobj = new First();
                firstobj.display();
        }
}
```

Output:

Hello from class First

Table 1.2.1 Access modifiers in Java

Access Modifier	Within Class	Within Package	Outside Package by Subclass Only	Outside Package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

The table 1.2.1 explains the visibility of class members (like variables or methods) based on the **access modifiers** in Java:

1. **Private**: Accessible only within the same class. Not accessible outside the class, even within the same package.
2. **Default**: Accessible within the same class and package. Not accessible in other packages, even by subclasses.
3. **Protected**: Accessible within the same class, package, and by subclasses in

other packages. Not accessible by non-subclass classes in other packages.

4. **Public**: Accessible from everywhere – within the same class, package, subclasses, and even outside the package.

This structure helps control the scope of access and ensures proper encapsulation in object-oriented programming.

1.2.5 Static Methods in java

In Java, static methods are those that can be invoked without needing to create an instance of the class. Instead of being tied to an object, static methods are associated with the class itself and can be accessed using the class name directly or by using a reference to an object of that class.

```
public class MyClass
{
    public static void geek(String name)
    {
        // Code to be executed
        System.out.println("Hello, " + name + "!");
    }
    public static int add(int a, int b)
    {
        return a + b;
    }
    public static float calculateArea(float radius)
    {
        return 3.14f * radius * radius;
    }
    public static void main(String[] args)
    {
        // Calling static methods
        geek("Alice");
        int sum = add(5, 10);
        float area = calculateArea(2.5f);
```

```
        System.out.println("Sum: " + sum);
        System.out.println("Area: " + area);
    }
```

Output:

Hello, Alice!

Sum: 15

Area: 19.625

Static methods are closely tied to the class they belong to and to call a static method, by using the syntax:

ClassName.methodName(arguments).

These methods are intended to be shared among all instances of the class, meaning they are not unique to any one object, but rather serve a common purpose for the class as a whole.

Another important aspect is that static methods cannot be overridden because they are resolved at compile time using static binding. Even if both the superclass and subclass declare a static method with the same name, the method in the subclass will hide the one in the superclass, a behavior known as method hiding, rather than true method overriding. In this case, when calling the method, the version associated with the class type (rather than the object) will be invoked. This distinction is important when designing class hierarchies that use static methods.

1.2.6 Final Keyword in java

The **final Keyword** in Java is used as a **non-access modifier** applicable only to a **variable**, a **method**, or a **class.** It is used to **restrict a user** in Java**.**

The following are **different contexts** where the final is used:

1. Variable
2. Method
3. Class

1.2.6.1 Final Variable

Final Variable is used to create a constant variable. When a variable is declared using the final keyword, its value cannot be altered, making it essentially a constant. This requires that the final variable be initialized at the time of declaration. If the final variable is a reference to an object, the reference itself cannot point to a different object, but the internal state of the object it references can still be modified. For example, you can add or remove elements from a final array or collection. As a convention, it is considered

good practice to write final variable names in uppercase letters, with words separated by underscores.

Syntax:

final <return type> <variable_name> = initialization;

Final variable Example Program

```
public class ConstantEx
{
        public static void main(String[] args)
        {
                // Define a constant variable PI
                final double PI = 3.14;
                // Display the value of PI
                System.out.println("Pi value is:" + PI);
        }
}
```

Output

Pi value is: 3.14

Different Ways to Use Final Variables in Java

1. Final Variable

A final variable is a constant that cannot be changed once assigned.

Example of Final variable declaration:

final int THRESHOLD = 5;

2. Blank Final Variable

A blank final variable is declared without initialization and must be assigned a value later, usually in the constructor.

Example of Blank final variable declaration:

final int THRESHOLD;

3. Static Final Variable

A static final variable is both constant and associated with the class rather than any instance. It is typically used for constant values shared across instances.

Example of Static final variable declaration:

static final double PI = 3.141592653589793;

4. Static Blank Final Variable

This is a blank final variable that is static and must be initialized within a static block. Example of Static blank final variable declaration:

static final double PI;

Initializing a Final Variable

A final variable must be initialized; otherwise, the compiler will throw an error. Once initialized, it cannot be changed. There are several ways to initialize a final variable:

- During Declaration:

This is the most common approach where the final variable is assigned a value at the time of declaration.

- In the Constructor or Instance-Initializer Block:

For a blank final variable, you must assign a value in the constructor or an instance-initializer block. If there are multiple constructors, the variable must be initialized in all of them, or a compile-time error will occur.

- In a Static Block:

A blank final static variable can be initialized in a static block to ensure that it is initialized before the class is loaded.

1.2.6.2 Final methods

Final Methods are used to prevent method overriding. When a method is declared with the final keyword, it is referred to as a final method. A final method cannot be overridden by any subclass. This ensures that the implementation of that method remains consistent across all subclasses and cannot be modified.

For instance, the Object class, which is the root of all classes in Java, contains several final methods that cannot be overridden. These methods provide essential functionality that Java developers rely on, so allowing them to be overridden would risk the integrity of the behavior expected from those methods.

Declaring a method as final is particularly useful when you want to prevent subclasses from changing the logic of that method. This ensures that the same implementation is preserved across all derived classes. For example, if you have a method in a superclass that performs a critical operation like security checks or resource handling, marking it as final guarantees that its behavior remains unaltered, safeguarding the program's functionality.

```
class SuperClass1
{
```

```
    public final void printing()
    {
        System.out.println("Final method example.");
    }
}
class SubClass extends SuperClass1
{
    // Attempting to override the final method will cause a compile-time error.
    public void printing()
    {
        System.out.println("overriding final method is not possible.");
    }
}
```

Output:

The code will produce a compile-time error.

In this example, the printing method is declared as final in the superclass SuperClass1. If you attempt to override it in the subclass SubClass, the compiler will throw an error.

Declaring methods as final, the method cannot be overridden, the Java compiler may perform certain optimizations like inlining, which can make the method execution faster. However, the primary reason for using final methods remains to preserve the intended behavior across the inheritance hierarchy.

1.2.6.3 Final Classes

Final Classes are used to prevent Inheritance. When a class is declared with the final keyword, it becomes a final class. A final class cannot be extended by any other class, meaning it cannot be inherited. This restriction ensures that the functionality and behavior of a final class remain unchanged and cannot be altered by subclassing.

Uses of Final Classes

Preventing Inheritance: The primary use of a final class is to prevent other classes from inheriting it. By declaring a class as final, you safeguard its implementation and prevent other developers from extending it and potentially modifying its behavior. This is useful when the design of the class is complete, and there is no need for further modification through inheritance. Many of Java's built-in wrapper classes, such as Integer, Float, Double, and Boolean, are final classes. These classes represent immutable objects and are designed to be used

as it is, So without the need for customization through inheritance.

Example:

```
final class Finaldemo1
{
   // methods and fields
}
// The following class declaration will cause a compile-time error
class Finaldemo2 extends Finaldemo1
{
   // COMPILE ERROR: Cannot extend final class A
}
```

In the above example, class Finaldemo1 is declared as final, which means class finaldemo2 cannot inherit from it. Any attempt to the class Finaldemo1 will result in a compile-time error, ensuring the integrity of Finaldemo1.

Creating Immutable Classes: Another important use of final classes is in creating immutable classes. An immutable class is one whose state cannot be changed after it is created. To ensure immutability, the class must be declared as final. If the class were not final, a subclass could potentially alter the state of the objects, breaking the immutability contract. A classic example of an immutable class is Java's built-in String class, which is declared as final. Once a String object is created, its value cannot be changed, ensuring that it remains constant throughout its lifecycle. By making the String class final, Java ensures that no subclass can override its behavior and modify its immutability.

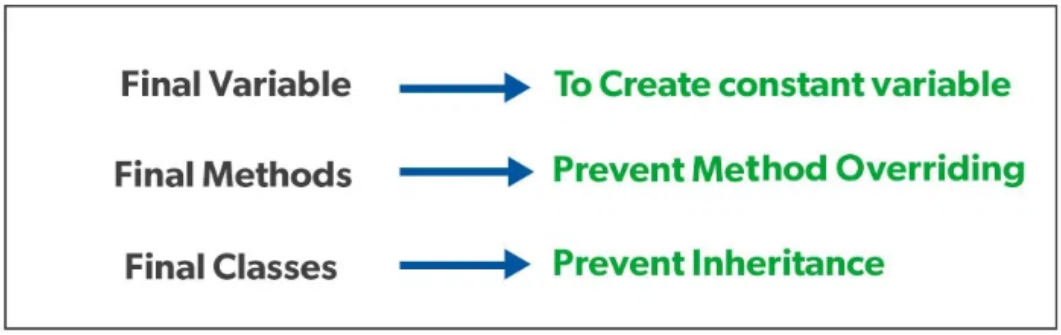
Example:

```
public final class ImmutableClass
{
        private final int value;
        public ImmutableClass(int value)
        {
                this.value = value;
         }
        public int getValue()
        {
```

```
        return value;
    }
}
```

In this example, ImmutableClass is declared as final to prevent inheritance, and its field value is declared as final to ensure it cannot be changed after the object is constructed. This is the fundamental approach used to create immutable objects in Java.

By using final classes, developers can create secure, immutable, and well-defined classes that cannot be altered or extended, ensuring better control over class behavior and integrity.



1.2.6.4 Characteristics of the final keywords in java

The final keyword serves to indicate that certain elements such as variables, methods, or classes, that cannot be modified or extended. Below are the key characteristics of the final keyword:

1. Final Variables: A variable declared as final can only be assigned a value once, making its value immutable after initialization. This is particularly useful for defining constants or values that must remain unchanged throughout the program.

2. Final Methods: When a method is marked as final, it cannot be overridden by subclasses. This is valuable when a method's behavior is critical to the class's functionality and must not be altered, ensuring consistency in the method's implementation.

3. Final Classes: Declaring a class as final means that it cannot be subclassed. This is used for classes that are intended to provide a complete, non-modifiable implementation, preventing inheritance from altering their structure or behavior.

4. Initialization of Final Variables: Final variables must be initialized either when they are declared or in the class constructor. This guarantees that they have a set value and cannot be changed after their initial assignment, ensuring data integrity.

5. Performance: The use of final can sometimes result in performance improvements. Since the compiler knows that certain variables or methods cannot change, it can apply optimizations, such as inlining method calls or reducing memory usage.

6. Security: The final keyword helps bolster security by ensuring that critical data or behavior cannot be modified, either accidentally or by malicious code, making it a useful tool for safeguarding sensitive parts of a program.

The final keyword enhances code stability, security, and maintainability by preventing unintended modifications to variables, methods, or classes. By using final, developers can write more robust, secure, and optimized code.

Recap

- Java is an Object-Oriented Programming (OOP) language, utilizing classes and objects to represent real-world entities.
- A class serves as a blueprint defining properties and behaviors common to all objects of that type.
- Objects are instances of classes, representing individual entities with unique attribute values.
- The concept of classes and objects allows for efficient code organization and scalability.
- A class encapsulates data (attributes) and methods (behaviors), promoting clarity and reusability in programming.
- Data members are variables that hold the state of an object, while methods define the actions the object can perform.
- Constructors are special methods that initialize an object's attributes when an instance is created.
- Nested classes enhance code organization by grouping related components logically within a parent class.
- Interfaces define a contract of methods that classes must implement, promoting flexibility in Java programming.
- Classes allow developers to create multiple instances with shared structures but unique data, adhering to OOP principles.
- Java methods are collections of statements that perform specific tasks and may return results.

- Methods must be defined within a class, making them essential for encapsulating object behavior in Java.
- The syntax of a method includes an access modifier, return type, method name, and optional parameters.
- Methods promote code reusability, allowing developers to call them multiple times throughout the program without redundancy.
- There are two main types of methods: predefined methods, which are built into Java's libraries, and user-defined methods, created by programmers for specific tasks.
- Instance methods are tied to specific objects, allowing them to access instance variables and be invoked after an object is created.
- Static methods are linked to the class and can be called without creating an object.
- The method signature includes the method's name and its parameters, but not the return type.
- Method names should start with a lowercase verb and use camel case for multiple words.
- Methods must be called explicitly to run, with instance methods requiring an object and static methods not.
- Methods can take different types of parameters, including arrays and variable arguments.
- Constructors initialize objects and must have the same name as the class, with no return type.
- There are three types of constructors: default (no parameters), parameterized (with parameters), and copy (duplicates another object).

Objective Type Questions

1. What is the blueprint or template used to define objects in Java?
2. What keyword is used to declare a class in Java?
3. What term describes an individual instance created from a class?
4. What keyword indicates that a variable's value cannot be changed?
5. What type of method is called when a new object is created?

6. What keyword is used to declare a method that cannot be overridden?
7. What is the return type for methods that do not return any value?
8. What access modifier allows a method to be accessed from any other class?
9. What term describes a class defined within another class?
10. What type of methods are built into Java's class libraries?
11. What type of method initializes the properties of a class?
12. What keyword is used to declare a variable that cannot be re-bound?
13. What type of method is associated with an instance of a class?
14. What access modifier restricts a method's visibility to the defining class only?
15. What is the primary benefit of using methods in programming?
16. What keyword is used to define a static method in Java?
17. What does the method signature include?
18. In Java, what naming convention is typically followed for method names?
19. What keyword must a constructor have the same name as?
20. How many types of constructors are there in Java?
21. Which access modifier restricts access only to the defining class?
22. What keyword is used to make a variable constant?
23. What does a copy constructor do?
24. Can a final class be inherited?
25. What is the primary benefit of using methods in Java?
26. What keyword prevents inheritance in Java?
27. Which Java class is an example of an immutable class?
28. What type of classes are the wrapper classes like Integer and Double?
29. What must be done to a final variable after initialization?
30. What does a final method prevent?

Answers to Objective Type Questions

1. Class
2. class
3. Object
4. final
5. Constructor
6. final
7. void
8. public
9. Nested
10. Predefined
11. Constructor
12. final
13. Instance
14. private
15. Reusability
16. static
17. Method name and parameters
18. Camel case
19. Class name
20. Three
21. Private
22. final
23. Copies data from another object

24. No

25. Reusability

26. final

27. String

28. final

29. Unchanged

30. Overriding

Assignments

1. Discuss the concept of classes and objects in Java, focusing on how classes act as blueprints for creating objects and the role of encapsulation in this structure.

2. Explain the components of a Java class, including data members, methods, constructors, nested classes, and interfaces, with examples to demonstrate their purpose in a class structure.

3. Describe the significance of methods in Java programming, including the syntax of a method declaration, the different types of methods, and the advantages they offer in terms of code reusability and optimization.

4. Define the four main access modifiers in Java and discuss their role in controlling visibility and accessibility with proper example programs.

5. Explain the role of constructors in Java, highlighting the differences between default, parameterized, and copy constructors. Include examples to illustrate how these constructors initialize object states and their key characteristics.

6. Describe the concept of static methods in Java, focusing on their declaration, calling, and significance compared to instance methods. Discuss the implications of method hiding and why static methods cannot be overridden.

7. Discuss the role of the final keyword in Java, focusing on its use in preventing inheritance and creating immutable classes.

References

1. "Effective Java" by Joshua Bloch Edition: 3rd Edition 2018 Addison-Wesl
2. "Effective Java" by Joshua Bloch, 3rd Edition, 2018, Addison-Wesley
3. "Head First Java" by Kathy Sierra and Bert Bates, 2nd Edition, 2005, O'Reilly Media
4. "Java Concurrency in Practice" by Brian Goetz, 1st Edition, 2006 Addison-Wesley
5. "Core Java Volume I – Fundamentals" by Cay S. Horstmann, 12th Edition, 2022, Pearson

Suggested Reading

1. Schildt, Herbert. "Java™ The Complete Reference Twelfth Edition." (2022).
2. Jana, Debasish. Java and object-oriented programming paradigm. PHI Learning Pvt. Ltd., 2005.
3. Baesens, Bart, Aimée Backiel, and Seppe Vanden Broucke. Beginning Java programming: the object-oriented approach. John Wiley & Sons, 2015.
4. Eckel, Bruce. Thinking in JAVA. Prentice Hall Professional, 2003.

Packages, I/O stream and Arrays

Learning Outcomes

The learner will be able to:

- identify the concept of packages in Java.
- define the purpose of import statements in Java.
- recognize the concept of arrays in Java.
- recall common operations performed on arrays.
- describe the purpose of input streams in Java.
- discuss the role of output streams in Java.

Prerequisites

As you dive deeper into Java programming, you may recall your earlier explorations of variables and control structures. These fundamental concepts are essential building blocks that lead you to more complex data management and manipulation techniques.

Imagine you have a treasure trove of data, much like a library filled with countless books. Each book represents a different piece of information, whether it be numbers, text, or images. But how do we keep track of all this data efficiently? This is where **arrays** come into play. Arrays allow you to store multiple values of the same type in a single variable, enabling you to manage collections of data easily.

Now, think about how you would communicate with the outside world. Just as you send letters or make phone calls to share information, your Java programs need to interact with external sources, like files or user inputs. This is where input/output (I/O) streams come into the picture. I/O streams allow your programs to read from and write to various data sources, making your applications dynamic and responsive.

Lastly, let's consider the concept of packages. Recall how organizing your notes into different folders makes studying easier. In Java, packages serve a similar purpose by grouping related classes and interfaces together. This organization not only streamlines your code but also enhances reusability and maintainability.

As you prepare to embark on this journey into packages, I/O streams, and arrays, think about the possibilities that await you. You'll unlock the power to manage data effectively, communicate seamlessly with users, and write organized, efficient code. Get ready to take your Java skills to the next level!

Keywords

Package, Import, InputStream, OutputStream, Array, Data Types, BufferedReader

Discussion

1.3.1 Java Packages

A Java package is a container for grouping related classes, interfaces, and sub-packages, helping developers manage and organize their code effectively. Java packages fall into two types: built-in packages and user-defined packages. Built-in packages, like java, lang, awt, javax, swing, net, io, util, and sql, are part of the Java standard library, while user-defined packages are created by developers for customized code organization.

Packages serve several purposes in Java:

1. Avoiding Naming Conflicts: Packages allow unique class names by grouping them under different package names.

2. Access Control: They regulate access levels for classes and interfaces.

3. Code Organization: They provide a structured, modular layout, especially for large projects with numerous classes.

1.3.2 Types of Java Packages

1.3.2.1 Built-in Packages

Java includes predefined packages that offer extensive functionality, including input/output processing, networking, and GUI development. Some key built-in packages are:

- java.sql: Provides classes for database access and processing, with classes like Connection, PreparedStatement, and ResultSet.

- java.lang: Contains core classes fundamental to Java, such as String, System, and Math.

- java.util: Includes collection classes and utilities, like ArrayList, HashMap, and Calendar.

- java.net: Provides classes for network applications, including Socket and URL.
- java.io: Supports I/O operations with classes such as BufferedReader, File, and PrintStream.
- java.awt: Contains classes for building graphical interfaces, with classes like Button, Font, and Graphics.

Fig. 1.3.1 shows Sample Java packages, subpackages and classes.

Example of Importing Built-in Packages. To use a class or package, you use the import statement:

Syntax:

```
import package.name.Class;   // Import a single class

import package.name.*;   // Import the whole package
```

Example

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

In the example above, java.util is a package, while Scanner is a class of the java.util package.

To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the nextLine() method, which is used to read a complete line:

Given below is an example using the Scanner class to get user input:

```
import java.util.Scanner;

class MyClass

{

        public static void main(String[] args)

        {

                Scanner myObj = new Scanner(System.in);

                System.out.println("Enter username");

                String userName = myObj.nextLine();

                System.out.println("Username is: " + userName);

        }

}
```

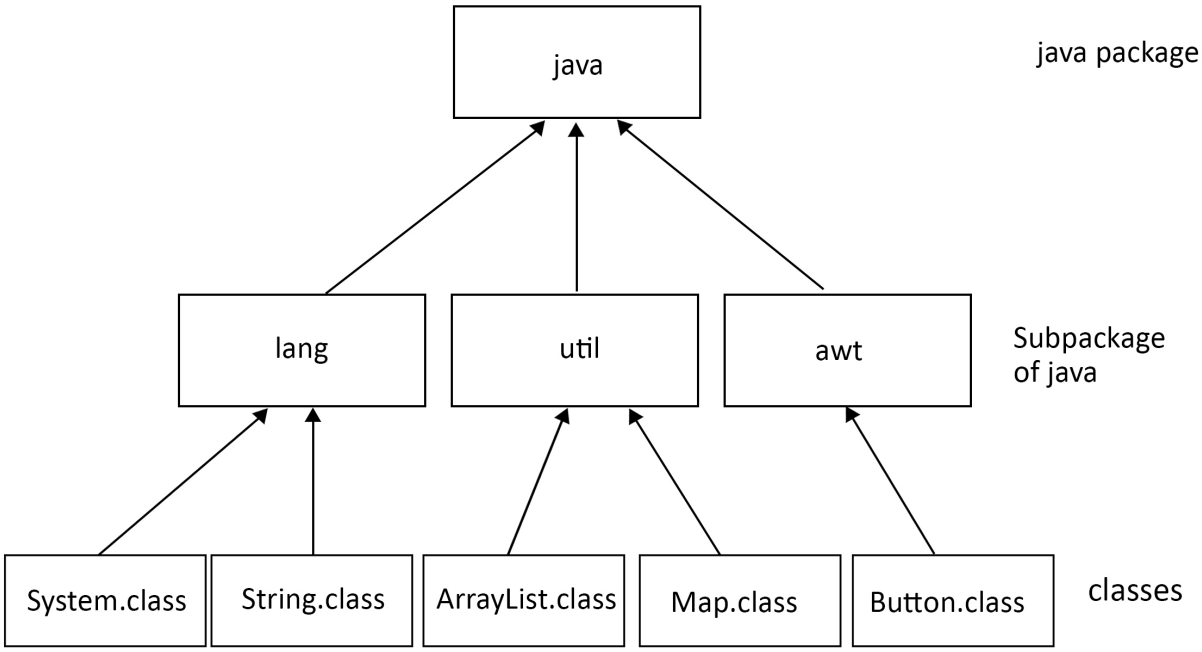


Fig. 1.3.1 Sample Java packages, subpackages and classes

1.3.2.2 User-Defined Packages

Java enables developers to create custom packages for better code modularity and reuse. These user-defined packages improve code organization and maintainability by grouping related classes together.

Benefits of User-Defined Packages

- Code Organization: Easier navigation of related code.
- Encapsulation: Restricts access levels within the package.
- Reusability: Packages allow sharing and reusing code across different projects.

Creating a User-Defined Package

1. Naming: Follow lowercase, reverse domain naming (e.g., com.example. myapp).
2. Declaration: Start each file with the package keyword.
3. File Structure: Arrange Java files according to the package hierarchy.

Example of user defined package

```
package com.example;

public class Calculator
```

```
{
        public int add(int a, int b)
        {
                return a + b;
        }
}
```

Using the User-Defined Package: To use classes from a user-defined package in another file

```
import com.example.Calculator;
public class PackageExample
{
        public static void main(String[] args)
        {
                Calculator calculator = new Calculator();
                System.out.println("Addition: " + calculator.add(5, 3));
        }
}
```

1.3.2.3 Accessing Classes from Different Packages

To access classes across different packages, you can:

1. Use package.*: Imports all classes within a package but excludes sub-packages.
2. Use package.classname: Imports a specific class from a package.
3. Use Fully Qualified Name: Refers directly to the package path without importing.

Example of Accessing Classes

```
// Accessing via package.*
import pack.*;
A obj = new A();
obj.msg();
// Accessing via package.classname
```

```
import pack.A;

A obj = new A();

obj.msg();

// Accessing via fully qualified name

pack.A obj = new pack.A();
```

1.3.2.4 Sub-Packages in Java

A sub-package is a package nested within another package, used for further categorization. For example, java.util has sub-packages like java.util.zip.

Example of Sub-Package Creation

```
package com.javapower.more;

class Simple

{

        public static void main(String args[])

        {

                System.out.println("Hello subpackage");

        }

}
```

1.3.2.5 Organizing Classes with Subpackages

Java lets you organize your code further by using subpackages. Imagine a package as a folder. A subpackage is like a subfolder inside that folder. This helps keep things tidy and easier to find.

For example, Java has a big "java" package with lots of classes for different tasks. To avoid clutter, Sun Microsystems (the original developers) created subpackages like "lang" (for language-related stuff), "net" (for networking), and "io" (for input/output).

This way, similar classes are grouped together. Networking classes like "Socket" and "ServerSocket" live in "net," while "Reader" and "Writer" for reading and writing data are in "io."

1.3.2.6 Naming Conventions

Packages and subpackages usually follow a naming pattern like domain.company.package. Think of it like an address: "com.javapower.more" or "org.ssit.dao."

Example: Creating a Subpackage

Let's create a subpackage called "more" inside a hypothetical "com.javapower" package.

Here's a simple class named "Simple" inside it:

```
package com.javapower.more;

public class Simple

{

        public static void main(String[] args)

        {

                System.out.println("Hello subpackage!");

        }

}
```

1.3.3 Import packages

In Java, the import statement is used to make classes or entire packages available within a Java program, allowing developers to access classes and interfaces from various packages without needing to type their fully qualified names repeatedly. Java provides built-in packages like java.util, java.io, and java.lang, the last of which is imported automatically in every program. By using import, developers can organize code more efficiently, accessing only the necessary classes or packages.

For example, import java.util.Scanner; allows a program to use the Scanner class directly to gather user input.

Using import package.name.*; includes all classes within a package, simplifying access when multiple classes from the same package are needed. This approach helps in structuring code more clearly and reducing redundancy.

Syntax

```
import package.name.Class;   // Import a single class

import package.name.*;   // Import the whole package
```

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1.3.3.1 Using packagename.*

If you use a package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//save by A.java
package pack;
public class A
{
	public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B
{
	public static void main(String args[])
	{
		A obj = new A();
		obj.msg();
	}
}
```

Output:Hello

1.3.3.2 Using packagename.classname

If you import package.classname then only the declared class of this package will be accessible.

Example of package by "import package.classname"

```
//save by A.java
package pack;
public class A
{
	public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.A;
class B
{
	public static void main(String args[])
	{
		A obj = new A();
		obj.msg();
	}
}
```

Output:Hello

1.3.3.3 Using fully qualified name

If you use a fully qualified name then only the declared class of this package will be accessible. Now there is no need to import. But you need to use a fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have the same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
//save by A.java
package pack;
public class A
{
	public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B
{
	public static void main(String args[])
```

```
    {
        pack.A obj = new pack.A();//using fully qualified name
        obj.msg();
    }
}
```

Output: Hello

If you import a package, subpackages will not be imported. If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Fig. 1.3.2 shows Sequence of the program must be package then import then class.

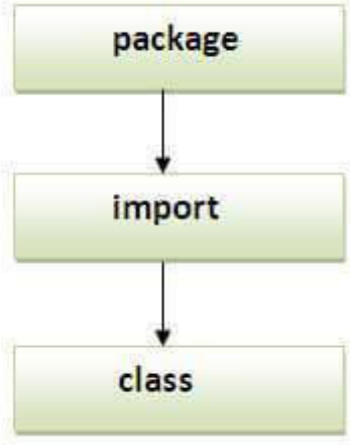


Fig. 1.3.2 Sequence of the program for importing subpackage

1.3.4 Default Java Package

Java automatically imports the java.lang package in every program. This package contains core classes required for basic Java functionality, such as String, System, and Math.

The java.lang package is automatically imported by the Java compiler, providing essential classes needed for creating basic Java programs. Among the most important classes in this package are Object, which serves as the root of all class hierarchies, and Class, whose instances represent classes at runtime.

Consider a simple Java program that checks if a number is even or odd. In this example, we do not explicitly import any packages. Despite this, the program can use the String class directly. Normally, the fully qualified class name is required at the beginning of a program using the import keyword, but the String belongs to java.lang package and the compiler handles the import implicitly.

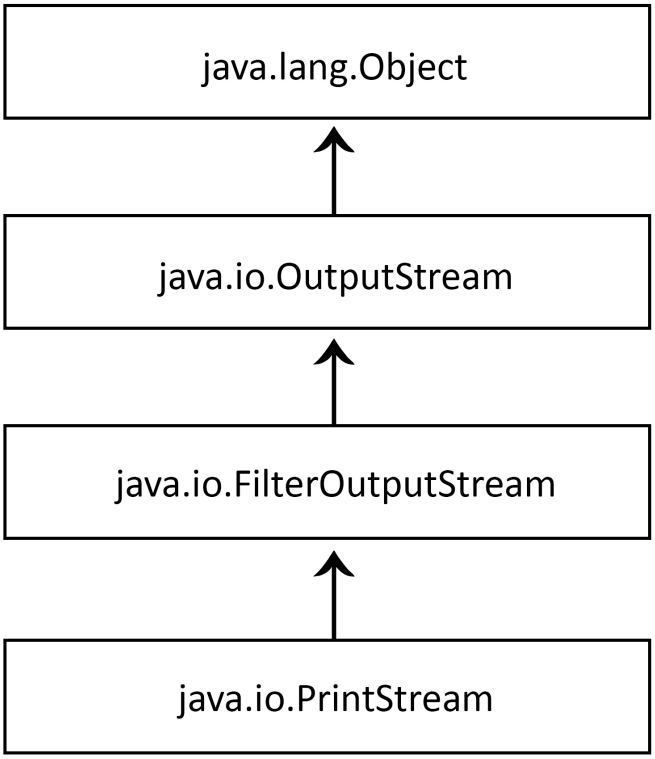


Fig 1.3.3 sample Java Builtin Packages and classes

By default, java.lang includes foundational classes like Object without any visible imports in the code. Fig. 1.3.3 shows sample Java Builtin Packages and class.

Example:

```
import java.io.PrintStream;

public class FindEvenOdd

{

        public FindEvenOdd() { }

        public static void main(String[] paramArrayOfString)

        {

                int i = 87;

                if (i % 2 == 0)

                {

                        System.out.println(i + " is an even number.");

                }

        else
```

```
            {
                System.out.println(i + " is an odd number.");
            }
        }
}
```

We do not need to explicitly import java.lang; all of its classes are accessible by default, simplifying program design and eliminating the need for additional import statements for this package.

By organizing classes into built-in and user-defined packages, Java allows for a more modular, manageable code structure, making applications easier to maintain and scale.

1.3.5 Java InputStream

The InputStream class in Java, located in the java.io package, is an abstract class that provides methods for reading bytes from various sources. It acts as the superclass for all byte input streams and implements the Closeable and AutoCloseable interfaces for efficient resource management. Essential methods include read(), which retrieves individual bytes or arrays of bytes, and close(), which frees system resources. This class simplifies input operations across files, network connections, and in-memory data, making it vital for Java I/O processes. For more information, you can refer to the official Java documentation. The InputStream class provides a simple and uniform way to access data, making it easier for developers to handle input operations without worrying about the details of the underlying data source. This class is essential when working with files, network connections, or even in-memory data.

One of the key features of InputStream is its ability to read data in a byte-oriented manner. This means that it can handle raw binary data efficiently. The class includes several methods for reading data, such as read(), which reads a single byte, and read(byte[] b), which reads multiple bytes into a byte array. These methods return the number of bytes read, allowing programs to process the data accordingly. The read() method will return -1 when the end of the stream is reached, which helps determine when to stop reading data.

InputStream is designed to work with various data sources, including files and network connections. For instance, the FileInputStream subclass reads data from files, while ByteArrayInputStream allows reading from byte arrays. This flexibility makes InputStream a powerful tool for developers, enabling them to create applications that can interact with different types of data sources seamlessly. By using the appropriate subclasses, developers can easily adapt their code to handle various input scenarios.

Error handling is also an important aspect when working with InputStream. It is common for input operations to encounter exceptions, such as IOException, which may occur due to issues like missing files or network problems. Therefore, developers need to implement proper error-handling techniques to ensure that their applications remain

robust and user-friendly. By catching these exceptions and providing meaningful error messages, developers can improve the overall user experience and maintain the reliability of their applications.

Java's InputStream is a fundamental component of the Java I/O system that simplifies data reading from various sources. Its ability to read bytes, work with different data types, and handle errors makes it an essential tool for Java developers. Understanding how to use InputStream effectively can lead to developing more efficient and reliable Java applications, as it provides a consistent way to manage input operations.

Java's InputStream class is extended by several subclasses, each designed for specific input sources. Key classes include:

- **FileInputStream**: Reads bytes from a file.
- **ByteArrayInputStream**: Reads bytes from a byte array in memory.
- **BufferedInputStream**: Buffers input to improve efficiency.
- **DataInputStream**: Reads Java primitive data types from an input stream.
- **PipedInputStream**: Implements an input stream connected to a piped output stream.

These subclasses allow flexible and efficient handling of various data sources.

Examples for common InputStream classes in Java is given below:

1. Example on FileInputStream:

```
import java.io.FileInputStream;
import java.io.IOException;
public class FileInputStreamExample
{
	public static void main(String[] args)
	{
		try (FileInputStream fis = new FileInputStream("example.txt"))
		{
			int data;
			while ((data = fis.read()) != -1)
			{
				System.out.print((char) data);
			}
```

```
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
        }
}
```

2. Example on ByteArrayInputStream:

```
import java.io.ByteArrayInputStream;
import java.io.IOException;
public class ByteArrayInputStreamExample
{
    public static void main(String[] args)
    {
        byte[] data = "Hello, ByteArray!".getBytes();
        try (ByteArrayInputStream bais = new ByteArrayInputStream(data))
        {
            int content;
            while ((content = bais.read()) != -1)
            {
                System.out.print((char) content);
            }
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

3. Example on BufferedInputStream:

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;
public class BufferedInputStreamExample
{
	public static void main(String[] args)
	{
		try (BufferedInputStream bis = new BufferedInputStream(new
		FileInputStream("example.txt")))
		{
			int data;
			while ((data = bis.read()) != -1)
			{
				System.out.print((char) data);
			}
		}
		catch (IOException e)
		{
			e.printStackTrace();
		}
	}
}
```

4. Example on DataInputStream:

```
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;
public class DataInputStreamExample
{
```

```
    public static void main(String[] args)
    {
        try (DataInputStream dis = new DataInputStream(new
        FileInputStream("data.bin")))
        {
            int intValue = dis.readInt();
            double doubleValue = dis.readDouble();
            System.out.println("Integer: " + intValue + ", Double: " +
            doubleValue);
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

5. Example on PipedInputStream:

```
import java.io.PipedInputStream;
import java.io.PipedOutputStream;
import java.io.IOException;
public class PipedInputStreamExample
{
   public static void main(String[] args)
  {
      try (PipedOutputStream pos = new PipedOutputStream();
      PipedInputStream pis = new PipedInputStream(pos))
     {
        new Thread(() ->
        {
           try
```

```
            {
                pos.write("Hello from PipedOutputStream!".getBytes());
            }
            catch (IOException e)
            {
               e.printStackTrace();
            }
        }).start();
            int data;
            while ((data = pis.read()) != -1)
            {
                System.out.print((char) data);
            }
        }
        catch (IOException e)
        {
           e.printStackTrace();
        }
    }
}
```

1.3.6 Java Output Stream

The OutputStream class in Java is an abstract class used for writing data to an output destination, such as a file or network socket. The OutputStream class in Java inherits several methods from the Object class, including clone(), equals(Object obj), hashCode(), toString(), and getClass(). This abstract class implements the interfaces Closeable and Flushable, which provide methods for closing the stream and flushing it, respectively. These features are essential for effective resource management and data integrity in applications that perform input and output operations. This class provides several important methods for writing data.

One of the key methods is close(), which releases system resources associated with the stream. This is essential for preventing memory leaks and ensuring that data is saved correctly. The flush() method plays a significant role in ensuring that any buffered output bytes are written out, maintaining data integrity during write operations.

The class also includes methods for writing bytes from byte arrays. The write(byte[] b) method writes an entire byte array to the stream, while the write(byte[] b, int off, int len) method allows for writing a specific portion of the array, starting from a specified offset. Additionally, the write(int b) method enables writing a single byte, giving programmers flexibility in managing their output.

It acts as a superclass for various output streams that allow programs to send data in a byte-oriented way. This standardization simplifies writing data to files, network connections, or other output locations. Using OutputStream helps developers create applications that require effective output handling.

The OutputStream class includes several important methods for data writing. One key method is write(int b), which allows for writing a single byte to the output stream. There is also the write(byte[] b) method, which writes an entire array of bytes at once. Additionally, a method exists for writing a part of a byte array, providing more control over the data sent. These methods do not return values but may throw exceptions if writing fails, which gives developers feedback on the operation's success.

Different subclasses of OutputStream offer specific functions for various output sources. For example, FileOutputStream writes data directly into files, while ByteArrayOutputStream writes data in a byte array in memory. This variety allows developers to switch between output destinations easily by using the right subclasses without changing much code. This flexibility is essential for building applications that work with different data sources.

Error handling is also important when using OutputStream. Input/output operations can lead to exceptions, such as IOException, due to problems like lack of storage space or network issues. Developers should implement error-handling techniques to keep applications stable and user-friendly. Catching these exceptions enables graceful handling of problems and provides helpful feedback to users, improving their overall experience with the application.

In summary, Java's OutputStream class is a key component for writing data to various output locations. Its byte-oriented design, along with different subclasses, allows developers to create applications that manage output operations effectively. Understanding how to use OutputStream can lead to more dependable Java applications, as it provides a consistent framework for handling output across different contexts.

Here are some classes that extend Java's OutputStream:

- **FileOutputStream**: Used to write bytes directly to a file.
- **ByteArrayOutputStream**: Writes bytes to a byte array in memory.
- **BufferedOutputStream**: Buffers output to improve performance.
- **DataOutputStream**: Writes Java's basic data types to an output stream.
- **PipedOutputStream**: Works with a piped input stream for inter-thread communication.

These classes help manage different types of output easily.

1.3.6.1 Example code using Java InputStream and OutputStream

An example code leveraging InputStream and OutputStream in Java using File InputStream and FileOutputStream is given below:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class StreamExample
{
    public static void main(String[] args)
   {
         try (InputStream input = new FileInputStream("input.txt");
         OutputStream output = new FileOutputStream("output.txt"))
         {
               int byteData;
               while ((byteData = input.read()) != -1)
               {
                     output.write(byteData);
               }
         }
         catch (IOException e)
         {
               e.printStackTrace();
         }
   }
}
```

1. Example on FileOutputStream

```
import java.io.FileOutputStream;
import java.io.IOException;
```

```
public class FileOutputStreamExample
{
    public static void main(String[] args)
    {
        try (FileOutputStream fos = new FileOutputStream("example.txt"))
        {
                String data = "Hello, FileOutputStream!";
                fos.write(data.getBytes());
                System.out.println("Data written to file.");
        }
        catch (IOException e)
        {
                e.printStackTrace();
        }
    }
}
```

2. Example on ByteArrayOutputStream

```
import java.io.ByteArrayOutputStream;
import java.io.IOException;
public class ByteArrayOutputStreamExample
{
    public static void main(String[] args)
    {
        try (ByteArrayOutputStream baos = new ByteArrayOutputStream())
        {
                String data = "Hello, ByteArrayOutputStream!";
                baos.write(data.getBytes());
                System.out.println("Data in byte array: " + baos.toString());
        }
```

```
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

3. Example on BufferedOutputStream

```
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class BufferedOutputStreamExample
{
    public static void main(String[] args)
    {
        try (BufferedOutputStream bos = new BufferedOutputStream(new
        FileOutputStream("buffered_example.txt")))
        {
            String data = "Hello, BufferedOutputStream!";
            bos.write(data.getBytes());
            bos.flush();
            System.out.println("Data written to buffered file.");
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

4. Example on DataOutputStream

```
import java.io.DataOutputStream;
```

```
import java.io.FileOutputStream;
import java.io.IOException;
public class DataOutputStreamExample
{
    public static void main(String[] args)
   {
            try (DataOutputStream dos = new DataOutputStream(new FileOutputStream
            ("data_example.txt")))
            {
                dos.writeUTF("Hello, DataOutputStream!");
                System.out.println("Data written to data file.");
             }
            catch (IOException e)
            {
                e.printStackTrace();
            }
    }
}
```

5. Example on PipedOutputStream :

```
import java.io.PipedOutputStream;
import java.io.PipedInputStream;
import java.io.IOException;
public class PipedOutputStreamExample
{
     public static void main(String[] args) throws IOException
     {
            PipedOutputStream pos = new PipedOutputStream();
            PipedInputStream pis = new PipedInputStream(pos);
            // Writing to PipedOutputStream in a separate thread
```

```
        new Thread(() ->
        {
            try
            {
                String data = "Hello, PipedOutputStream!";
                pos.write(data.getBytes());
                pos.close();
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
        }).start();
            // Reading from PipedInputStream
             byte[] buffer = new byte[50];
            int bytesRead = pis.read(buffer);
            System.out.println("Data read: " + new String(buffer, 0, bytesRead));
    }
}
```

These examples demonstrate how to use various OutputStream classes to write data in different contexts. In short,

1. FileOutputStream: Writes bytes to a file, creating or overwriting it if it already exists.
2. ByteArrayOutputStream: Collects bytes in memory, allowing easy access and conversion to a string.
3. BufferedOutputStream: Buffers data for efficient writing to an output stream, reducing the number of writes.
4. DataOutputStream: Writes Java primitive data types to an output stream, preserving their binary format.
5. PipedOutputStream: Sends data to a connected PipedInputStream, enabling inter-thread communication.

1.3.7 Arrays

Suppose you're organizing a bookshelf where each shelf can only hold one type of book, say all novels, all biographies, or all science books. This is like an array—an indexed collection of similar, or homogeneous, data elements. One of the best aspects of having an organized bookshelf is the ability to store a large collection of books in one location, making them easy to locate and access. Similarly, in programming, arrays help you store multiple values under a single variable. It improves the clarity and readability of your code.

However, there's a downside. Once you've built your bookshelf, it's fixed in size. If you made space for 100 books, but later needed room for 110, you're stuck. You can't expand or shrink the bookshelf to fit your changing needs. This limitation mirrors the challenge with arrays: once an array is created, its size is locked in place. This is tricky. Just like you don't always know how many books you'll need to store, you might not know how much data your array will need to hold in advance.

Definition: An array is a collection of the same type of data which is stored in contiguous memory location. In Java, an array is an **object** which contains elements of a similar data type.

An array can contain primitive data types (int, char, etc.) and non-primitive data types(object references of a class) depending on the definition of the array.

1.3.7.1 Array Declaration

As in other programming languages like C,C++,etc , we have to declare variables in Java before it is used in the program.An array declaration has two components: the type and the name.

There are three different syntaxes for declaring an array in Java.They are :

1. int[ ] x;
2. int [ ]x;
3. int x[ ];

All the three syntaxes are valid for declaration of variables in Java. This statement declares an array named x which holds integers. Like an array of integers, we can also create an array of other primitive data types like char, float, double, etc., or user-defined data types (objects of a class). For example:

```
double[ ] y;  // array of double data type
MyClass myClassArray[ ];    // array of Object
```

One thing you have to understand is that the size of the array is specified only at creation not during declaration.

1.3.7.2 Array Creation

Every array in java is an object.Hence we can create arrays using 'new' operator. The

syntax is as shown below.

int[ ] x = new int[3];

where 'x' is the reference variable.

1.3.7.3 Access an element in an array

We can access array elements using their index, which starts from 0. For an integer array x with *five elements*, x[0] represents the first element, x[1] represents the second element, x[2] represents the third element and so on. The last index will be 4 in this case.

1.3.7.4 Array Initialization

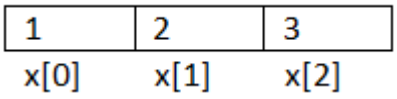
int[ ] x = new int[3]; // Array creation

x[0]=1; // Assigning value 1 to x[0]

x[1]=2; // Assigning value 2 to x[1]

x[2]=3; // Assigning value 3 to x[2]

The array after these statements are executed is shown below



1.3.8 Array Declaration,Creation and Initialization in a single statement

Now, think about a suitcase that can only hold whole numbers, like 10, 20, and 30. In programming, you'd declare, create, and initialize this suitcase of numbers in a single step. It's as if you're saying, *"I need a suitcase for numbers, and here's what I'll pack in it right now."*

int[ ] x = {10,20,30};

Here, int is the type of item the suitcase (array) will hold—whole numbers. The [ ] indicates that this is a suitcase (an array), not just a single item. And inside the curly braces, {10, 20, 30}, you've packed the numbers right from the start.

Some other examples :

char[ ] ch = {'a','e','i','o','u'};

String[ ] s = {"A","AA","AAA"};

1.3.9 Length of Array

We can get the length of an array using the length property.Length of the array means

that the total number of elements the array can hold. length is a final variable applicable for arrays.

eg: int[ ] x = new int[6];

System.out.println(x.length); // prints the value 6

1.3.10 Multidimensional Arrays

Multidimensional arrays can be described simply as arrays that contain other arrays. The data in these arrays is organized in a tabular structure, where elements are stored in row-major order.

Example : Two dimensional array:

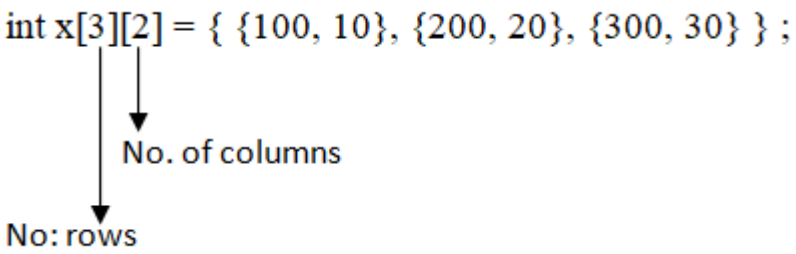
int[ ][ ] x = new int[10][20];

Three dimensional array:

int[ ][ ][ ] y = new int[10][20][30];

How can you initialize a 2-D array?

As in the case of 1-D arrays, you can provide values for a 2-D at the time of declaration. The values can be given in braces as in the example given below.



The above 2-D array will be stored in the memory as shown below:

100	10
200	20
300	30

The values in the array can be accessed using indices. x[0][0] is value 100, x[0][1] is value 10, x[1][0] is value 200 and so on.

1.3.11 Accessing array elements using for - loop

Each element in the array can be accessed using its index, which starts at 0 and goes up

to array length - 1. To access and display all elements of the array ‘x’, a for loop can be used:

```
for (int i = 0; i < x.length; i++)

        System.out.println("Element at index " + i + " : "+ x[i]);  // prints all the elements in the array
```

Enhanced for loop / Java for-each loop

The for-each loop, also called the enhanced for loop, was introduced in Java with J2SE 5.0. It offers a simplified way to iterate through arrays or collections. Instead of traditional loops, it allows for direct traversal of each element, improving code readability and reducing the chances of errors. The name "for-each" reflects how the loop processes elements one at a time in sequence.

Imagine you have a stack of books on your desk and you need to review each one. In a traditional for loop, you would go through the stack by counting the books, starting at the first book (index 0), and moving on until you reach the last one (index n-1). However, with a for-each loop, it's like having an assistant taking and giving you each book one by one, without you having to keep track of how many books are there or which one you’re on. You simply review the book given to you and move on to the next. This process is simpler and less prone to mistakes, just like how the for-each loop eliminates the need for managing index values and ensures each element is accessed directly.

The syntax of Java for-each loop consists of data_type with the variable followed by a colon (:), then array or collection.

```
for (data_type variable : array / collection)

{

        //body of for-each loop

}
```

For eg:

(1) int[ ] x = { 10, 20,30,40};

To print elements of the above array:

Normal loop

```
for(int i = 0 ; i < x.length ;  i++)

System.out.println(x[i]);
```

Enhanced for loop

```
for(int x1 : x)

System.out.println(x1);
```

(2) For 2-D arrays

Normal loop

```
for(int i = 0;  i<x.length ;  i++)
{
        for (int j=0 ; j<x[i].length ;  j++)
        {
                System.out.println(x[i]);
        }
}
```

Enhanced for loop

```
for(int x1 : x)
{
        for( int x2: x1)
        {
                System.out.println(x2);
        }
}
```

The enhanced for loop executes only in sequence.ie the counter is always increased by one , whereas in normal for loop you can change the steps as per your wish. eg: doing something like i= i+2;

You can read values into the array by using **scanner class**. The Scanner class in Java is used to easily read input from various sources such as user input from the keyboard, files, or other data streams. It simplifies the process of breaking input into tokens and supports reading different data types like integers, strings, and floating-point numbers. This makes it highly useful for interactive console-based programs.

1.3.12 Example programs

1.3.12.1 Program No:1

Write a Java program to read and print 10 integers.

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

```
public class ReadAndPrintIntegers
{
    public static void main(String[ ] args)
    {
        // Create a scanner object to read input
        Scanner scanner = new Scanner(System.in);
         // Create an array to store 10 integer values
        int[  ] numbers = new int[10];
        // Prompt user to enter 10 integers
        System.out.println("Enter 10 integer values:");
        // Loop to read integers into the array
        for (int i = 0; i < 10; i++)
        {
                numbers[i] = scanner.nextInt();
        }
         // Print the integers entered by the user
         System.out.println("You entered the following values:");
         for (int i = 0; i < 10; i++)
        {
                System.out.println(numbers[i]);
        }
        // Close the scanner
        scanner.close( );
     }
}
```

OUTPUT

Enter 10 integer values:

12 45 78 22 36 49 57 81 92 11

You entered the following values:

12 45 78 22 36 49 57 81 92 11

Calling scanner.close( ) is **not strictly necessary** in small programs, but it is considered a good practice to free up resources and avoid potential issues in larger applications.

1.3.12.2 Program No : 2

Write a Java program to find the smallest element in an array.

```
import java.util.Scanner;
public class SmallestElement {
    public static void main(String[ ] args) {
        // Create a scanner object to read input
        Scanner scanner = new Scanner(System.in);
        // Prompt the user to enter the size of the array
        System.out.print("Enter the number of elements in the array: ");
        int n = scanner.nextInt( );
        // Create an array to store the elements
        int[ ] numbers = new int[n];
        // Prompt the user to enter the array elements
        System.out.println("Enter " + n + " integer values:");
        for (int i = 0; i < n; i++) {
            numbers[i] = scanner.nextInt( );
        }
        // Initialize the smallest element as the first element in the array
        int smallest = numbers[0];
        // Loop to find the smallest element
        for (int i = 1; i < n; i++) {
            if (numbers[i] < smallest) {
                smallest = numbers[i];
            }
        }
        // Print the smallest element
```

```
        System.out.println("The smallest element in the array is: " + smallest);
        // Close the scanner
        scanner.close( );
    }
}
```

OUTPUT

```
Enter the number of elements in the array: 5
Enter 5 integer values:
10
3
15
7
1
The smallest element in the array is: 1
```

1.3.12.3 Program No : 3

Write a java program to read and print the values in a 2-D array.

```
import java.util.Scanner;
public class TwoDArrayInput {
    public static void main(String[ ] args) {
        // Create a Scanner object to read input
        Scanner scanner = new Scanner(System.in);
        // Prompt the user to enter the size of the array
        System.out.print("Enter the number of rows: ");
        int rows = scanner.nextInt( );
        System.out.print("Enter the number of columns: ");
        int cols = scanner.nextInt( );
        // Create a 2-D array with the specified size
        int[ ][ ] array = new int[rows][cols];
        // Loop to read input into the 2-D array
        System.out.println("Enter the elements of the 2-D array:");
        for (int i = 0; i < rows; i++) {
```

```
            for (int j = 0; j < cols; j++) {
                System.out.print("Element at [" + i + "][" + j + "]: ");
                array[i][j] = scanner.nextInt( );
            }
        }
        // Print the 2-D array
        System.out.println("\nThe elements of the 2-D array are:");
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                System.out.print(array[i][j] + " ");
            }
            System.out.println( );  // New line after each row
        }
        // Close th e scanner
        scanner.close( );
    }
}
```

OUTPUT

```
Enter the number of rows: 2
Enter the number of columns: 3
Enter the elements of the 2-D array:
Element at [0][0]: 1
Element at [0][1]: 2
Element at [0][2]: 3
Element at [1][0]: 4
Element at [1][1]: 5
Element at [1][2]: 6
The elements of the 2-D array are:
1 2 3
4 5 6
```

Recap

- Java packages group related classes and interfaces, making code more organized and avoiding name conflicts. There are built-in packages (like java.lang and java.util) and user-defined ones created by developers.
- Built-in packages handle common functions like input/output, networking, and graphics. User-defined packages help structure projects and reuse code.
- The import statement brings classes from packages into a program. You can import specific classes, all classes in a package, or use the full class name without importing.
- Sub-packages organize code further by creating nested packages, like java.util.zip within java.util, to keep code easy to find.
- Java automatically includes java.lang, which has core classes like String and Object, so you don't need to import it yourself.

Java Input Stream

- Package: IntputStream belongs to the java.io package.
- Definition: InputStream is an abstract class in Java that allows reading bytes from various sources, such as files and network connections.
- Key Methods:
  - read(): Reads a single byte or an array of bytes.
  - close(): Closes the stream and releases associated resources.
- Interfaces: Implements Closeable and AutoCloseable for efficient resource management.
- Subclasses: Various subclasses, like FileInputStream and BufferedInputStream, provide specific implementations for different input sources.
- Use Cases: Commonly used for file operations, data streaming, and communication between applications.

Java Output Stream

- Package: OutputStream belongs to the java.io package.
- Definition: OutputStream is an abstract class in Java for writing bytes to various output destinations, such as files or network connections.
- Key Methods:
  - write(int b): Writes a single byte to the output stream.
  - write(byte[] b): Writes an array of bytes to the stream.

- flush(): Forces any buffered output bytes to be written.
- close(): Closes the stream and releases resources.

- Interfaces: Implements Closeable and Flushable for effective resource management.
- Subclasses: Includes FileOutputStream, BufferedOutputStream, and PrintStream for specific output needs.
- Use Cases: Widely used for file writing, data transmission, and communication between applications.
- Arrays are indexed collections of similar data types.
- Array size is fixed upon creation and cannot be changed.
- An array in Java is an object that holds elements of a specific type.
- Arrays can store both primitive and non-primitive data types.
- Declaration syntax includes the data type followed by brackets (e.g., int[] x).
- Arrays can be initialized in a single statement (e.g., int[] x = {10, 20, 30}).
- The length of an array can be accessed using the length property.
- Multidimensional arrays consist of arrays within arrays (e.g., 2D or 3D arrays).
- The enhanced for loop simplifies iteration through array elements.
- The Scanner class can be used to read input values into arrays.

Objective Type Questions

1. What type of package is created by developers to improve code organization and reuse?
2. Which package in Java is automatically imported by the compiler?
3. Which package in Java provides classes for network applications?
4. What keyword is used to declare a package in Java?
5. In which package is the Scanner class found?
6. Which package provides classes for database access?

7. What is the purpose of a sub-package in Java?
8. What is the primary function of the InputStream class?
9. Which method reads a single byte from an InputStream?
10. What does the close() method do in an OutputStream?
11. Which interface allows the OutputStream to flush its data?
12. What value does the read() method return when the end of a stream is reached?
13. What method is used to write a byte array to an OutputStream?
14. Which method in OutputStream forces any buffered output bytes to be written out?
15. What is the purpose of the write(int b) method in OutputStream?
16. What does the write(byte[] b, int off, int len) method do?
17. Which class is the parent of both InputStream and OutputStream?
18. What is an array in Java?
19. How do you declare an array to hold integers in Java?
20. What is the index of the last element in an array of size 5?
21. How can you initialize an array in a single statement?
22. What property is used to get the length of an array in Java?
23. What does a multidimensional array in Java contain?
24. What loop can be used to iterate through each element of an array in Java?
25. Which class in Java is commonly used to read input from the user?

Answers to Objective Type Questions

1. User-defined
2. java.lang

3. java.net
4. package
5. java.util
6. java.sql
7. Categorization
8. InputStream reads bytes from a source.
9. The method read() reads a single byte.
10. close() releases resources in OutputStream.
11. The Flushable interface enables flushing.
12. read() returns -1 at the end of a stream.
13. write(byte[] b) writes a byte array.
14. flush() forces buffered bytes to write out.
15. write(int b) writes a single byte.
16. write(byte[] b, int off, int len) writes specific bytes from an array.
17. Both InputStream and OutputStream extend java.io.InputStream.
18. A collection of similar data.
19. int[] x;
20. 4
21. int[] x = {10, 20, 30};
22. Using the length property.
23. Other arrays.
24. A for loop.
25. Scanner class.

Assignments

1. Explain the concept of Java packages and their significance in code organization. Describe the purpose of Java packages, the types of packages (built-in and user-defined), and how they help avoid naming conflicts and regulate access control. Give examples to illustrate how packages improve code organization and maintainability in large projects.

2. Discuss the use and benefits of user-defined packages in Java. Explain how to create a user-defined package, including the naming conventions and file structure. Describe the benefits of user-defined packages in terms of code reusability, encapsulation, and organization. Provide a sample code to demonstrate the creation and usage of a user-defined package.

3. Analyze the role of the java.lang package in Java programming. Describe the significance of the java.lang package and list some key classes within it, such as Object, String, and System. Explain why this package is automatically imported in every Java program and discuss how its classes are essential for basic Java functionalities. Provide an example demonstrating the use of a java.lang class without an explicit import statement.

4. Write a Java program that initializes an array of 10 integers with user-defined values. Calculate and display the average of the numbers in the array.

5. Develop a Java program that prompts the user to enter 5 numbers, stores them in an array, and then finds and displays the maximum and minimum values from the array.

6. Write a Java program that initializes an array with 6 elements. Create a method to reverse the elements of the array and print the reversed array to the console.

References

1. "Effective Java" by Joshua Bloch Edition: 3rd Edition 2018 Addison-Wesl

2. "Effective Java" by Joshua Bloch, 3rd Edition, 2018, Addison-Wesley

3. "Head First Java" by Kathy Sierra and Bert Bates, 2nd Edition, 2005, O'Reilly Media

4. "Java Concurrency in Practice" by Brian Goetz, 1st Edition, 2006

5. Addison-Wesley

6. "Core Java Volume I – Fundamentals" by Cay S. Horstmann, 12th Edition, 2022, Pearson

Suggested Reading

1. Schildt, Herbert. "Java™ The Complete Reference Twelfth Edition." (2022).

2. Jana, Debasish. Java and object-oriented programming paradigm. PHI Learning Pvt. Ltd., 2005.

3. Baesens, Bart, Aimée Backiel, and Seppe Vanden Broucke. Beginning Java programming: the object-oriented approach. John Wiley & Sons, 2015.

4. Eckel, Bruce. Thinking in JAVA. Prentice Hall Professional, 2003.

Abstraction Inheritance Overriding and Overloading

Learning Outcomes

The learner will be able to:

- understand the concept of abstraction and how it helps in hiding the internal details of objects while exposing only the necessary functionalities.
- differentiate between abstract classes and interfaces and know when to use each in Java.
- familiarise the concept of inheritance and how it allows a subclass to acquire the properties and behaviors of a superclass.
- the concept of method overriding and how it allows a subclass to provide a specific implementation of a method defined in its parent class.
- understand the concept of method overloading, where multiple methods can have the same name but differ in the number or type of parameters.

Prerequisites

When you drive a car, the essential controls like the **steering wheel**, **pedals**, and **gear shift** are exposed to you. You use these controls to drive the car without needing to understand the internal working of the **engine**, **transmission**, or **fuel injection system**. These complex details are hidden (abstracted) from you, allowing you to interact with the car using a simple interface. In Java, complex internal operations and programming logic are hidden from the user, while only the necessary and relevant functionality is exposed. This concept is known as **abstraction**.

Consider a "Vehicle" class with properties like speed and capacity. If we need to create a new class named "Car," it can **inherit** the common properties of the Vehicle class while also having its own unique properties, such as numberOfDoors or fuelType. This demonstrates the concept of **inheritance** in Java, where a subclass adopts the common features of its parent class and can extend them with its own characteristics. This promotes code reuse and establishes a hierarchical relationship between classes.

In Java, **method overriding** allows a subclass to provide its own implementation of a

method that is already defined in its parent class. For example, the Vehicle class may have a method called displayInfo() to show general details like speed and capacity. The Car class, which inherits from Vehicle, can override this method to add more specific details, such as the number of doors. This way, when the displayInfo() method is called on a Car object, it displays both the inherited and car-specific details, demonstrating how overriding customizes or enhances inherited behavior.

In Java, **method overloading** allows a class to have multiple methods with the same name but different parameter lists. For example, in the Vehicle class, we could have a method named displayInfo(). One version of this method could take no parameters and display basic information like speed and capacity. Another version could take additional parameters, such as the vehicle's model or color, and display more specific details. Method overloading is determined at compile-time, allowing flexibility in how methods are called based on the provided arguments. This helps improve code readability and usability while maintaining the same method name for related functionality.

Keywords

Class, interface, extends, polymorphism, abstract method

Discussion

In Java, object-oriented programming relies on key concepts like abstraction, inheritance, overriding, and overloading to create flexible and reusable code. Abstraction hides implementation details and focuses on essential features, allowing developers to work with high-level interfaces. Inheritance enables a subclass to inherit properties and methods from a superclass, promoting code reuse. Overriding lets a subclass redefine an inherited method to provide specific functionality, while overloading allows multiple methods with the same name but different parameters to coexist, enhancing flexibility and readability. These concepts work together to simplify code design and maintenance.

1.4.1 Abstraction in Java Programming

In Java, complex internal operations and programming logic are hidden from the user, while only the necessary and relevant functionality is exposed. This concept is known as **abstraction**. Abstraction allows users to interact with an object without needing to understand its intricate details or how it works internally. For example, when you use a car, you don't need to know how the engine processes fuel or how the transmission operates. All you care about are the essential features like accelerating, braking, or steering, and the car handles the complex operations behind the scenes. Similarly, in Java programming, abstraction allows you to focus on what an object does, rather than how it does it.

This facility makes programs easier to use, as it hides unnecessary complexities, and

provides a cleaner, more user-friendly interface. Abstraction is achieved through the use of **abstract classes** and **interfaces**, which specify what operations an object can perform without revealing how those operations are implemented. By employing abstraction, developers can build more modular, maintainable, and flexible systems, since internal implementations can change without affecting the user-facing interface. This separation of concerns leads to simpler, cleaner code that is easier to manage and extend.

In Java, abstraction can be achieved in two ways:

1. Abstract classes
2. Interfaces

1.4.1.1 Abstract Classes

An abstract class in Java acts as a partially implemented class that itself cannot be instantiated. It exists only for subclassing purposes, and provides a template for its subcategories to follow. Abstract classes can have implementations with abstract methods, that is methods without implementation and non-abstract methods, methods with bodies and can implement. Abstract methods are declared to have no body, leaving their implementation to subclasses. An abstract is a Java modifier applicable for classes and methods in Java but not for Variables. Use them when you have a base class with common code (shared behavior) that should not be instantiated on its own.

1.4.1.1.1 Key characteristics:

- An abstract class must be declared with an abstract keyword.
- An abstract class can have both abstract and non-abstract methods.
- It can have member variables, constructors, and concrete methods.
- It can have static methods.
- It can have final methods which will force the subclass not to change the body of the method.
- The subclass that extends an abstract class must provide implementations for all the abstract methods of the abstract class.

1.4.1.1.2 Benefits of Abstraction:

- **Security**: Hides implementation details and only exposes functionality.
- **Code Reusability**: Abstract classes and interfaces can be reused by multiple classes.
- **Loose Coupling**: By hiding the details of how an object works, you allow flexibility in changing the implementation without affecting the user of the object.

1.4.1.1.3 Abstract method

A method that is declared as abstract and does not have implementation is known as

abstract method.

Syntax for abstract class

```
abstract class class name
 {
        abstract void methodname(); // abstract method
        public void methodname() // non-abstract method
        {
                //Body of the function
        }
 }
```

Example for an abstract class

The program below demonstrates the concept of abstract classes and methods in Java. It defines an abstract class Shape with an abstract method draw() and a concrete method description(). A subclass Circle is created to provide the implementation of the draw() method. The subclass is instantiated, and both the draw() and description() methods are called to illustrate the functionality.

```
abstract class Shape
{
        abstract void draw(); // Abstract method
        public void description() // Non-abstract method
        {
                 System.out.println("This is a shape.");
        }
}
class Circle extends Shape
 {
         void draw()              // Provide implementation of the abstract method
        {
                System.out.println("Drawing a circle.");
         }
}
```

```
public class AbstractClassExample
{
        public static void main(String[] args)
        {
                Shape shape = new Circle();   // Create an object of the subclass
                shape.draw();                 // Call the methods
                shape.description();
        }
}
```

Output

Drawing a circle.

This is a shape.

In this example, Shape is an abstract class with one abstract method draw() and one concrete method description(). Subclasses of Shape must implement the draw() method, but they can inherit the description() method.

1.4.1.1.4 Example Questions

Write a program in Java to demonstrate the concept of an abstract class and method. Define an abstract class Animal with an abstract method makeSound() and a concrete method eat(). Create two subclasses, Dog and Cat, that provide their own implementations of the makeSound() method. Instantiate objects of both subclasses and call their methods to show the behavior.

```
// Abstract class
abstract class Animal
{
        abstract void makeSound();          // Abstract method
        public void eat()                   // Concrete method
        {
                System.out.println("This animal eats food.");
        }
}
class Dog extends Animal                    // Subclass 1: Dog
{
```

```
        void makeSound()
        {
                System.out.println("The dog barks: Woof Woof!");
        }
}
class Cat extends Animal // Subclass 2: Cat
{
        void makeSound()
        {
                System.out.println("The cat meows: Meow Meow!");
        }
}
public class AbstractClassExample2
{
        public static void main(String[] args)
        {
                Animal dog = new Dog();      // Create objects of Dog and Cat
                Animal cat = new Cat();
                dog.makeSound();              // Call methods on Dog
                dog.eat();
                cat.makeSound();              // Call methods on Cat
                cat.eat();
         }
}
```

Output

```
The dog barks: Woof Woof!
This animal eats food.
The cat meows: Meow Meow!
This animal eats food.
```

1.4.1.2 Interfaces in java

An interface in Java is a blueprint for a class. It contains only abstract methods and default/ static methods. Interfaces are used to achieve abstraction and multiple inheritance in Java, as a class can implement multiple interfaces. There can be only abstract methods in the java interface, not method body. Java Interface also represents IS-A relationship. It cannot be instantiated just like abstract class. Use them when you want to define a contract or a set of methods that various unrelated classes can implement.

Key Features of Java Interfaces

- An interface in Java is a completely abstract class.
- It can contain only abstract methods , interfaces can also have default and static methods with implementations.
- An interface provides a blueprint for classes, and a class that implements an interface must implement all of its methods.
- A class can implement multiple interfaces, allowing Java to achieve multiple inheritance in a way.
- Methods in an interface are implicitly public and abstract
- Variables in an interface are implicitly public, static, and final.

Syntax for interface

```
interface interfacName
{
        void method name();              // abstract method by default
}
```

An example for interface

Below program to demonstrate the use of an interface. Define an interface Drawable with method draw(). Create two classes, Circle and Rectangle, that implement the Drawable interface. Instantiate objects of these classes and call the methods.

```
interface Drawable
{
        void draw();    // abstract method by default
}
class Circle implements Drawable
{
      public void draw()
      {
```

```
            System.out.println("Drawing a circle.");
        }
}
class Rectangle implements Drawable
{
    public void draw()
    {
            System.out.println("Drawing a rectangle.");
     }
}
public class TestInterface
{
      public static void main(String[] args)
      {
            Drawable shape1 = new Circle();
            shape1.draw();      // Drawing a circle
            Drawable shape2 = new Rectangle();
            shape2.draw();      // Drawing a rectangle
       }
}
```

1.4.1.2.1 Example Question

Write a program to demonstrate the use of an interface. Define an interface Vehicle with methods start() and stop(). Create two classes, Car and Bike, that implement the Vehicle interface. Instantiate objects of these classes and call the methods.

```
interface Vehicle        // Define the interface
{
      void start();        // Abstract method
       void stop();        // Abstract method
}
class Car implements Vehicle            // Implementing the interface in Car class
{
     @Override
      public void start()
```

```
    {
        System.out.println("The car starts with a key.");
    }
    @Override
    public void stop()
    {
        System.out.println("The car stops by applying brakes.");
    }
}
class Bike implements Vehicle          // Implementing the interface in Bike class
{
    @Override
    public void start()
    {
        System.out.println("The bike starts with a kick.");
    }
    @Override
    public void stop()
    {
        System.out.println("The bike stops by applying brakes.");
    }
}
public class InterfaceExample
{
    public static void main(String[] args)
    {
        Vehicle myCar = new Car();          // Create objects of Car and Bike
        Vehicle myBike = new Bike();
        myCar.start();                      // Call methods on Car
        myCar.stop();
        myBike.start();                     // Call methods on Bike
        myBike.stop();
```

```
    }
}
```

Expected Output:

The car starts with a key.

The car stops by applying brakes.

The bike starts with a kick.

The bike stops by applying brakes.

1.4.1.2.2 Relationship Between Classes and Interfaces

As shown in the figure1.4.1 given below, a class extends another class, an interface extends another interface, but a class implements an interface.

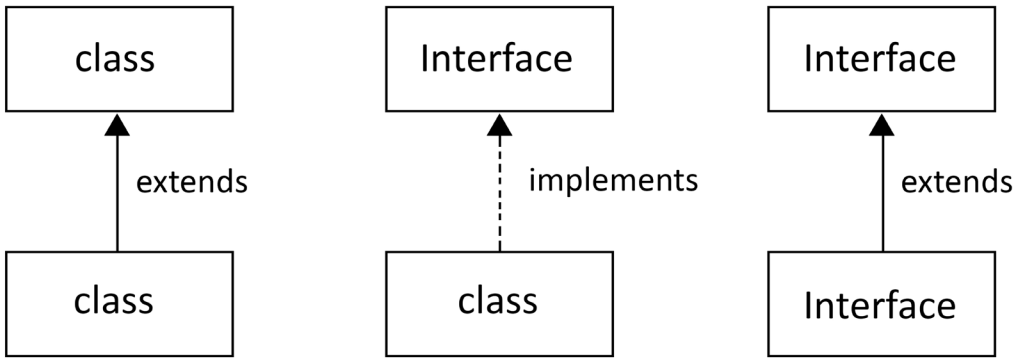


Fig 1.4.1 Relation between class and interface

1.4.1.3 Key Differences Between Abstract Class and Interface:

Abstract Class	Interface
Can have abstract and concrete methods.	Contains only abstract methods.
Can have member variables and constructors.	Cannot have member variables or constructors.
A class can extend only one abstract class.	A class can implement multiple interfaces.
Can have access modifiers for methods.	Methods are public by default in an interface.
Suitable when classes share common behavior.	Suitable for declaring common functionality across unrelated classes.

1.4.2 Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP), including Java. It allows one class (called a **child** or **subclass**) to inherit properties and behaviors (fields and methods) from another class (called a **parent** or **superclass**). Inheritance promotes code reuse and establishes a relationship between the child and parent class, forming a hierarchy. Inheritance in Java allows for the creation of a natural hierarchy, enabling code reuse, logical organization, and the development of flexible and maintainable applications. By inheriting properties and methods from parent classes, subclasses become specialized versions of those parents, and they can modify or extend behaviors to suit their specific needs.

1.4.2.1 Key Features of Inheritance:

1. **Code Reusability**: The child class can reuse methods and properties defined in the parent class, reducing code duplication.

2. **Method Overriding**: The child class can provide its own specific implementation of a method that is already defined in the parent class.

3. **Polymorphism**: Inheritance enables the use of polymorphism, where a parent class reference can point to a child class object.

4. **Parent-Child Relationship**: It creates a logical relationship between classes. The child class is a specialized version of the parent class.

1.4.2.1.1 Terms used in Inheritance

Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

Reusability: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

Syntax of inheritance:

```
class Parent
{
        public void showMessage()    // Parent class members (fields and methods)
        {
                System.out.println("This is a message from the parent class.");
```

```
        }
}
class Child extends Parent
{
    // Child class inherits properties and methods from Parent
    public void showMessage()
    {
        System.out.println("This is a message from the child class.");
    }
}
public class TestInheritance
{
    public static void main(String[] args)
    {
        Child child = new Child();
        child.showMessage();  // Calls the overridden method in the child class
    }
}
```

1.4.2.1.2 Example for inheritance

Below code demonstrate inheritance in java

```
class Animal
{
    void eat()
    {
        System.out.println("This animal eats food.");
    }
}
class Dog extends Animal                // Child class
{
```

```
    void bark()
    {
        System.out.println("The dog barks.");
    }
}
public class InheritanceDemo          // Main class
{
    public static void main(String[] args)
    {
        Dog dog = new Dog();    // Create an object of the child class
        // Call methods from the parent class and child class
        dog.eat();              // Inherited from the Animal class
        dog.bark();             // Defined in the Dog class
    }
}
```

Output

This animal eats food.

The dog barks.

The Parent Class (Animal) contains the eat method, which prints a message. The Child Class (Dog) inherits the eat method from the Animal class and adds its own bark method. In the Main Method, an instance of the Dog class is created, and both the inherited eat method and the bark method are called. This demonstrates how a child class can utilize both its own methods and the methods inherited from the parent class.

1.4.2.2 Types of Inheritance in Java:

On the basis of class, there can be three types of inheritance in java:

- Single inheritance
- Multilevel Inheritance
- Hierarchical Inheritance

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

1.4.2.2.1 Single Inheritance:

A child class inherits from one parent class. Fig 1.4.2 shows single inheritance

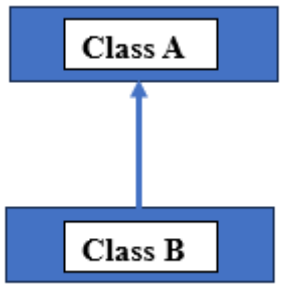


Fig 1.4.2 Single inheritance

Example: Car inherits from Vehicle.

When the Car class inherits from the Vehicle class, it means that the Car class gains access to the fields and methods defined in the Vehicle class. For example, if the Vehicle class has a method like start() or a property like speed, the Car class can use them without redefining them. Additionally, the Car class can have its own unique methods and properties, such as drive() or fuelType. This inheritance relationship represents an "is-a" relationship, meaning a Car is a type of Vehicle. This allows for code reuse and a clear hierarchical structure, where common features are defined in the parent class (Vehicle), and specific features are added in the child class (Car).

```
class Vehicle
{
    void start()
    {
        System.out.println("Vehicle is starting.");
    }
}
class Car extends Vehicle      // Child class
{
    void drive()               // The Car class inherits the start() method from Vehicle
    {
        System.out.println("Car is driving.");
    }
}
```

```
public class InheritanceExample                    // Main class
{
        public static void main(String[] args)
        {
                Car myCar = new Car();        // Create an object of the Car class
                 myCar.start();              // Call the inherited start() method from Vehicle
                 myCar.drive();              // Call the drive() method specific to Car
        }
}
```

Output:

Vehicle is starting.

Car is driving.

1.4.2.2.2 Multilevel Inheritance:

A child class inherits from a parent class, and that parent class inherits from another parent class, forming a chain. Figure 1.1.3 shows the structure of Multilevel inheritance.

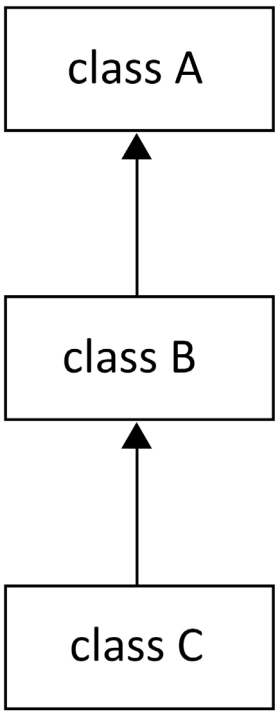


Fig 1.4.3 Multilevel Inheritance

Example: ElectricCar inherits from Car, which in turn inherits from Vehicle.

The ElectricCar class inherits from the Car class, which itself inherits from the Vehicle class. This creates a multilevel inheritance hierarchy where ElectricCar gains access to methods and properties from both Car and Vehicle. For example, if Vehicle has a start() method and Car adds a drive() method, ElectricCar can use both while also defining its own features, such as chargeBattery(). This structure showcases how inheritance allows for progressive addition of functionality across levels.

```
class Vehicle
{
    void start()
    {
        System.out.println("Vehicle is starting.");
    }
}
// Child class that inherits from Vehicle
class Car extends Vehicle
{
    void accelerate()
    {
        System.out.println("Car is accelerating.");
    }
}
// Grandchild class that inherits from Car
class ElectricCar extends Car
{
    void charge()
    {
        System.out.println("Electric car is charging.");
    }
}
// Main class to demonstrate multilevel inheritance
public class InheritanceDemo
```

```
{
    public static void main(String[] args)
    {
        // Create an object of ElectricCar
        ElectricCar myElectricCar = new ElectricCar();
        // Call methods from all classes in the hierarchy
        myElectricCar.start();          // Inherited from Vehicle
        myElectricCar.accelerate();     // Inherited from Car
        myElectricCar.charge();         // Defined in ElectricCar
    }
}
```

Output

Vehicle is starting.

Car is accelerating.

Electric car is charging.

1.4.2.2.3 Hierarchical Inheritance:

Multiple child classes inherit from the same parent class. Figure 1.1.4 shows the structure of hierarchical inheritance.

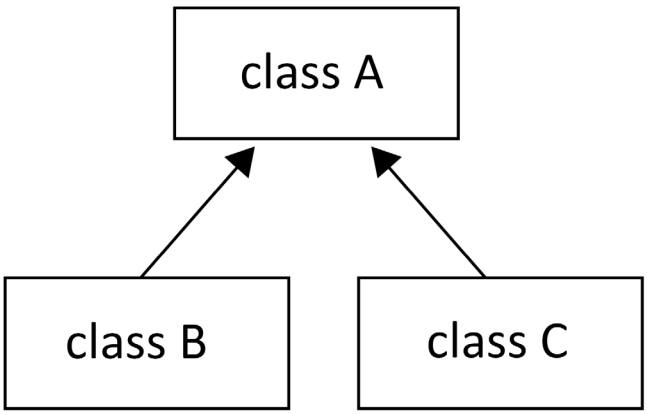


Fig 1.4.4 Heirarchical Inheritance

Example: Both Car and Bike inherit from Vehicle.

In the case of hierarchical inheritance, both the Car and Bike classes inherit from the common Vehicle class. This means that Car and Bike gain access to the methods and

properties defined in the Vehicle class, such as a start() method. Each subclass can then add its own specific features, like accelerate() in Car or ride() in Bike, while still sharing the common functionality provided by the Vehicle class. This structure demonstrates how multiple classes can inherit from a single parent class, promoting code reuse and maintaining a logical hierarchy.

```
class Vehicle
{
    void start()
    {
        System.out.println("Vehicle is starting.");
    }
}
// Child class Car inherits from Vehicle
class Car extends Vehicle
{
    void drive()
    {
        System.out.println("Car is driving.");
    }
}
// Child class Bike inherits from Vehicle
class Bike extends Vehicle
{
   void ride()
   {
        System.out.println("Bike is riding.");
   }
 }
// Main class to demonstrate hierarchical inheritance
public class InheritanceDemo
{
```

```
    public static void main(String[] args)
        {
            // Create objects of Car and Bike
            Car myCar = new Car();
            Bike myBike = new Bike();
            // Call methods from the Vehicle, Car, and Bike classes
            myCar.start();    // Inherited from Vehicle
            myCar.drive();    // Defined in Car
            myBike.start();   // Inherited from Vehicle
            myBike.ride();    // Defined in Bike
        }
}
```

Output:

Vehicle is starting.

Car is driving.

Vehicle is starting.

Bike is riding.

1.4.2.2.4 Java Does Not Support Multiple Inheritance:

Java **does not** support multiple inheritance using classes, meaning a class cannot inherit from more than one parent class. This avoids ambiguity issues that arise from having multiple parents with the same method signatures. However, Java allows **multiple inheritance** through **interfaces**.

```
interface Drivable
{
    void drive();
}
interface Flyable
{
    void fly();
}
```

```
class FlyingCar implements Drivable, Flyable
{
    public void drive()
    {
        System.out.println("Flying car is driving.");
    }
    public void fly()
    {
        System.out.println("Flying car is flying.");
    }
}
```

1.4.2.2.5 Benefits of Inheritance:

1. **Reusability**: Child classes reuse the fields and methods of the parent class, reducing code duplication.
2. **Maintainability**: Changes made in the parent class automatically reflect in child classes, making maintenance easier.
3. **Extensibility**: Child classes can extend or enhance the functionality of parent classes.
4. **Polymorphism**: Inheritance enables polymorphic behavior, where objects of different classes can be treated as objects of a common parent class.

1.4.3 Method overriding

Method overriding in Java occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The method in the child class must have the same name, return type, and parameters as the method in the parent class. Method overriding in Java allows subclasses to provide a specific implementation of a method that is already defined in a parent class. It is a core feature of OOP that supports runtime polymorphism and allows flexible and extensible code design.

1.4.3.1 Key Points of Method Overriding:

1. **Same Method Signature**: The method in the child class must have the same name, return type, and parameter list as the method in the parent class.
2. **Inheritance**: Method overriding occurs only in the context of inheritance. The child class inherits the method from the parent class and overrides it with its own implementation.

3. **Access Modifier**: The access level of the overriding method cannot be more restrictive than the method it overrides. For example, if the parent method is public, the overridden method must also be public.

Annotations: The @Override annotation can be used to indicate that a method is being overridden. This is optional but helps in code clarity and error detection.

Runtime Polymorphism: Method overriding is used to achieve **runtime polymorphism**, where the method that is executed is determined at runtime based on the object type.

Syntax of Method Overriding:

```
class Parent
{
   // Method to be overridden
     public void showMessage()
    {
          System.out.println("Message from Parent class.");
     }
 }
class Child extends Parent
{
   // Overriding the method of Parent class  @Override
      public void showMessage()
     {
          System.out.println("Message from Child class.");
      }
 }
public class TestOverriding
{
        public static void main(String[] args)
        {
                Parent parent = new Parent();
                 parent.showMessage();  // Output: Message from Parent class.
```

```
        Child child = new Child();
        child.showMessage();   // Output: Message from Child class.
        Parent parentChild = new Child();
        parentChild.showMessage();  // Output: Message from Child class
    }
}
```

1.4.3.2 Example of Overriding in a Real-World Scenario:

Consider a **Vehicle** class with a method startEngine() and its subclasses like **Car** and **Bike**. Each type of vehicle starts the engine differently, so the startEngine() method will be overridden in each subclass.

```
class Vehicle  // Parent class
{
    public void startEngine()        // Method to be overridden
    {
        System.out.println("Vehicle is starting.");
    }
}
class Car extends Vehicle       // Child class Car overriding startEngine method
{
    public void startEngine()  @Override
    {
        System.out.println("Car engine starts with a key.");
    }
}
class Bike extends Vehicle  // Child class Bike overriding startEngine method
{
    public void startEngine()          // @Override
    {
        System.out.println("Bike engine starts with a kick.");
    }
```

```
}
public class TestVehicles
{
    public static void main(String[] args)
    {
        Vehicle myCar = new Car();
        myCar.startEngine();          // Output: Car engine starts with a key.
        Vehicle myBike = new Bike();
        myBike.startEngine();       // Output: Bike engine starts with a kick.
    }
}
```

1.4.3.3 Rules for Method Overriding:

1. **Return Type**: The return type must be the same or a subclass (covariant return type) of the method in the parent class.

2. **Method Signature**: The method signature (method name, parameter types, and number of parameters) must be identical to the method in the parent class.

3. **Access Level**: The overriding method cannot reduce the visibility of the inherited method. For example:

   - If the parent method is public, the overriding method must also be public.
   - If the parent method is protected, the overriding method can be protected or public, but not private.

4. **Exceptions**: The child class cannot throw a broader exception than the parent class method. It can throw the same exceptions, or more specific exceptions (narrowed exceptions).

5. **Static Methods**: Static methods cannot be overridden. If a subclass defines a static method with the same signature as a static method in the parent class, this is known as method hiding, not overriding.

6. **Final Methods**: Methods declared as final in the parent class cannot be overridden in the subclass.

7. **Private Methods**: Private methods in the parent class cannot be overridden because they are not visible to the child class.

1.4.3.4 @Override Annotation:

The @Override annotation is used to explicitly indicate that a method is overriding a method in the superclass. It is not mandatory but is highly recommended as it helps in two ways:

- It makes the code clearer by explicitly indicating that a method is overridden.
- It helps the compiler catch errors. If a method signature doesn't exactly match the method in the parent class, the compiler will generate an error.

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

1.4.3.5 Runtime Polymorphism and Method Overriding

One of the primary uses of method overriding is to achieve **runtime polymorphism**. In Java, a parent class reference can refer to a child class object. When an overridden method is called using this reference, the method that is executed is determined at runtime based on the actual object, not the reference type.

```
Parent obj = new Child();
obj.show();  // Will call the Child's show() method due to runtime polymorphism.
```

1.4.3.6 Why Use Method Overriding?

1. **Dynamic Method Dispatch**: It helps Java support dynamic method dispatch,

where the method called is determined at runtime based on the object.

2. **Code Flexibility**: By overriding methods, a subclass can provide its specific behavior, while still maintaining the same interface as the parent class.

3. **Reuse Parent Class Code**: The child class can inherit and extend the functionality of the parent class without having to rewrite all the code from scratch.

1.4.4 Method Overloading

Method overloading in Java is a feature that allows a class to have more than one method with the same name, but with different parameter lists. The difference can be in the number of parameters, types of parameters, or both. Method overloading is a form of **compile-time polymorphism** because the decision about which method to call is made at compile time. Method overloading is a useful feature in Java that allows the same method name to handle different types or numbers of arguments, improving code readability and flexibility. It's a form of compile-time polymorphism, as the method to be invoked is determined during compilation based on the method signature.

1.4.4.1 Key Points of Method Overloading:

1. **Same Method Name, Different Parameters**: The overloaded methods must have the same name but different parameter lists.

2. **Return Type Doesn't Matter**: Method overloading is determined by the method's parameter list, not by its return type. You cannot overload methods based on return type alone.

3. **Compile-Time Polymorphism**: The compiler determines which method to invoke based on the method signature (name + parameters) at compile time.

1.4.4.2 Ways to Overload a Method:

1. **Different Number of Parameters**: Methods can have the same name but differ in the number of parameters.

2. **Different Types of Parameters**: Methods can have the same name and number of parameters but differ in parameter types.

3. **Different Order of Parameters**: Methods can have the same name and parameters but differ in the order of parameters (if they have different data types).

Example for Method Overloading:

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

```
        {
                return a + b;
         }
         public int add(int a, int b, int c)        // Overloaded method to add three integers
        {
                return a + b + c;
        }
                // Overloaded method to add two double values
        public double add(double a, double b)
        {
                return a + b;
         }
}
public class TestOverloading
{
        public static void main(String[] args)
        {
            Calculator calc = new Calculator();
           System.out.println(calc.add(5, 10)); // Calls with two int parameters
           System.out.println(calc.add(5, 10, 15)); // Calls with three int parameters
           System.out.println(calc.add(5.5, 10.2)); // Calls with two double parameters
      }
}
```

Output:

15

30

15.7

1.4.4.3 Examples of Method Overloading:

a. Overloading by Changing the Number of Parameters:

Below shows example of method overloading, the Display class defines two show() methods with the same name but different parameters. The first method takes a single integer a as an argument and prints it, while the second method takes two integers a and b and prints both. In the TestDisplay class, the show() method is called twice: once with a single integer (5) and once with two integers (10 and 20). This demonstrates how method overloading allows multiple methods with the same name to be differentiated by the number of parameters, enabling more flexible and reusable code.

```
class Display
{
        public void show(int a)
        {
              System.out.println("Integer: " + a);
        }
        public void show(int a, int b)
        {
              System.out.println("Two Integers: " + a + ", " + b);
         }
}
public class TestDisplay
{
         public static void main(String[] args)
         {
                  Display display = new Display();
                  display.show(5);                  // Calls show(int a)
                  display.show(10, 20);             // Calls show(int a, int b)
          }
}
```

b. Overloading by Changing the Parameter Type:

Below example of method overloading, the Display class defines two show() methods with the same name but different parameter types. The first method accepts an integer (int a) and prints it, while the second method accepts a string (String s) and prints it. In

the TestDisplay class, the show() method is called with both an integer (5) and a string ("Hello"), demonstrating how method overloading allows methods to have the same name but be differentiated by their parameter types. This enhances the flexibility of the code by enabling multiple methods to handle different data types.

```
class Display
{
        public void show(int a)
        {
                System.out.println("Integer: " + a);
        }
        public void show(String s)
        {
                System.out.println("String: " + s);
         }
 }
public class TestDisplay
{
        public static void main(String[] args)
        {
                Display display = new Display();
                display.show(5);          // Calls show(int a)
                display.show("Hello");    // Calls show(String s)
        }
}
```

c. Overloading by Changing the Order of Parameters:

In this example of method overloading, the Display class defines two show() methods with the same name but different parameter orders. The first method takes a String followed by an int, while the second method takes an int followed by a String. In the TestDisplay class, the show() method is called twice: first with a string ("Hello") and an integer (10), which calls the method with the String, int parameter order, and second with an integer (20) and a string ("World"), which calls the method with the int, String parameter order. This demonstrates how method overloading allows the same method

name to be used with parameters in different orders, providing flexibility in how the methods are called.

```
class Display
{
        public void show(String s, int a)
        {
                System.out.println("String: " + s + ", Integer: " + a);
         }
        public void show(int a, String s)
        {
                System.out.println("Integer: " + a + ", String: " + s);
         }
}
public class TestDisplay
{
        public static void main(String[] args)
        {
                Display display = new Display();
                display.show("Hello", 10);              // Calls show(String, int)
                display.show(20, "World");              // Calls show(int, String)
         }
}
```

1.4.4.4 Rules of Method Overloading:

Different Parameter List: Methods must differ in the number or type of parameters.

Cannot Overload by Return Type Alone: You cannot overload methods based only on return type. The compiler won't be able to differentiate between two methods with the same name and parameters but different return types.

Example of invalid overloading:

```
public int add(int a, int b)
 { ... }
```

public double add(int a, int b) { ... } // Compilation error

Access Modifiers and Exception Handling: Changing the access modifier or throwing different exceptions in overloaded methods does not constitute method overloading. It must differ based on the parameter list.

1.4.4.5 Method Overloading with Type Promotion:

Java allows **automatic type promotion** when no exact match is found for a method signature. For example, if an int method is called and only double methods are available, the int value will be promoted to double.

```
class Display
{
        public void show(double a)
        {
                System.out.println("Double: " + a);
         }
}
public class TestDisplay
{
        public static void main(String[] args)
        {
                Display display = new Display();
                display.show(10);        // int value is promoted to double, and double a
        }
}
```

In this case, Java will promote the int to double because the method signature with double matches.

1.4.4.6 Benefits of Method Overloading:

1. **Code Readability**: It makes code more readable and clean since similar functionality can be handled using the same method name with different parameters.
2. **Reusability**: It allows methods to be reused with different input types or numbers of arguments, reducing the need to create separate methods for similar tasks.

3. **Compile-Time Polymorphism**: Overloading allows the compiler to resolve which method to call based on the parameters at compile time, leading to more flexible code.

1.4.4.7 Example of method overloading :

Consider an application that needs to print different types of data, such as integers, doubles, or strings. Instead of creating different method names (printInt, printDouble, printString), you can use method overloading.

```
class Printer
{
        public void print(int value)
        {
                System.out.println("Printing Integer: " + value);
        }
        public void print(double value)
        {
                System.out.println("Printing Double: " + value);
         }
        public void print(String value)
        {
                System.out.println("Printing String: " + value);
        }
}
public class TestPrinter
{
        public static void main(String[] args)
        {
                Printer printer = new Printer();
                printer.print(10);              // Prints an integer
                printer.print(12.34);           // Prints a double
                printer.print("Hello");          // Prints a string
        }
}
```

Recap

- Abstraction in Java Programming
- Abstract Classes
  - Key characteristics
  - Benefits of Abstraction
  - Abstract method
- Interfaces in java
- Key Differences Between Abstract Class and Interface:
- Inheritance
  - Key Features of Inheritance
  - Types of Inheritance in Java
  - Single Inheritance
  - Multilevel Inheritance
  - Hierarchical Inheritance
  - Java Does Not Support Multiple Inheritance
  - Benefits of Inheritance
  - Example of Inheritance
- Method overriding
  - Key Points of Method Overriding
  - Example of Overriding in a Real-World Scenario
  - Rules for Method Overriding
  - @Override Annotation
  - Runtime Polymorphism and Method Overriding
  - Why Use Method Overriding?
- Method Overloading
- Key Points of Method Overloading
- Ways to Overload a Method

- Examples of Method Overloading
- Rules of Method Overloading
- Method Overloading with Type Promotion
- Benefits of Method Overloading
- Example of method overriding

Objective Type Questions

1. What is abstraction in object-oriented programming?
2. Which of the following best describes inheritance?
3. What is the purpose of method overriding?
4. Which keyword is used in Java to create a class that cannot be instantiated directly but can only be subclassed?
5. Which of the following statements about method overloading is correct?
6. In which of the following cases would method overriding be used?
7. Which of the following is NOT true about inheritance?
8. What happens if you attempt to overload methods by only changing the return type in Java?
9. In which scenario would you use method overloading?
10. What is the key difference between method overloading and method overriding?

Answers to Objective Type Questions

1. Abstraction allows exposing only essential features while hiding implementation details
2. Inheritance allows a class to use methods from another class.
3. To extend or alter the behavior of a method in a child class.

4. Abstract

5. Method overloading occurs when multiple methods in the same class share the same name but differ in the number or type of parameters.

6. When a subclass needs to implement a method with the same signature as one in its parent class but with different functionality.

7. Inheritance is always transitive; a class can inherit from multiple parent classes.

8. The program will not compile because method signatures must differ in more than just the return type.

9. When you need to define the same method with the same name but different numbers or types of parameters.

10. Overloading refers to methods with the same name but different signatures, while overriding refers to redefining a method in a subclass.

Assignments

1. Design an abstract class BankAccount that contains abstract methods like deposit(), withdraw(), and checkBalance(). Implement concrete classes like SavingsAccount and CurrentAccount that extend the BankAccount class and provide implementations for the abstract methods.

2. Write a short essay explaining how abstraction helps manage complexity in large-scale software development and how it promotes maintainability and scalability.

3. Create a base class Vehicle that has properties like speed, fuelType, and numWheels. Then, create subclasses like Car, Truck, and Motorcycle that inherit from Vehicle and have additional properties or methods unique to each subclass.

4. Model a real-world scenario using multilevel inheritance. For example, a Person class can be the base class, followed by a Student class inheriting from Person, and then a GraduateStudent class inheriting from Student. Implement methods that are specific to each class in the hierarchy.

5. Write a program that has a base class Animal with a method makeSound(), and two derived classes Dog and Cat that override makeSound(). Demonstrate polymorphism by calling the makeSound() method on objects of Dog and Cat through a reference to Animal.

6. Create a base class Employee with a constructor that initializes basic information. Then, create a derived class Manager that overrides the constructor and adds more information. Explain how constructor chaining works when inheritance is involved.

7. Create a Calculator class with multiple add() methods. Overload the add() method to handle different parameter types (integers, doubles, and multiple arguments). Write a test case to demonstrate the use of each overloaded method.

8. Design a class FileHandler that contains overloaded methods readFile(). Overload this method to handle text files, CSV files, and JSON files. Show how the method signatures differ while keeping the method name the same.

9. Create an abstract class Payment with methods like processPayment(). Then, implement classes like CreditCardPayment, PaypalPayment, and BankTransferPayment that inherit from Payment and override processPayment(). Use method overloading in one of the payment methods to handle different types of payment amounts (e.g., single payment vs. installment payments).

10. Create a base class Shape with an overloaded method calculateArea() for different shapes (circle, square, and rectangle).

11. Then, override the calculateArea() method in derived classes like Circle, Square, and Rectangle, and show how method overloading and overriding work together in the program.

References

1. "Effective Java" by Joshua Bloch Edition: 3rd Edition 2018 Addison-Wesl

2. "Effective Java" by Joshua Bloch, 3rd Edition, 2018, Addison-Wesley

3. "Head First Java" by Kathy Sierra and Bert Bates, 2nd Edition, 2005, O'Reilly Media

4. "Java Concurrency in Practice" by Brian Goetz, 1st Edition, 2006 Addison-Wesley

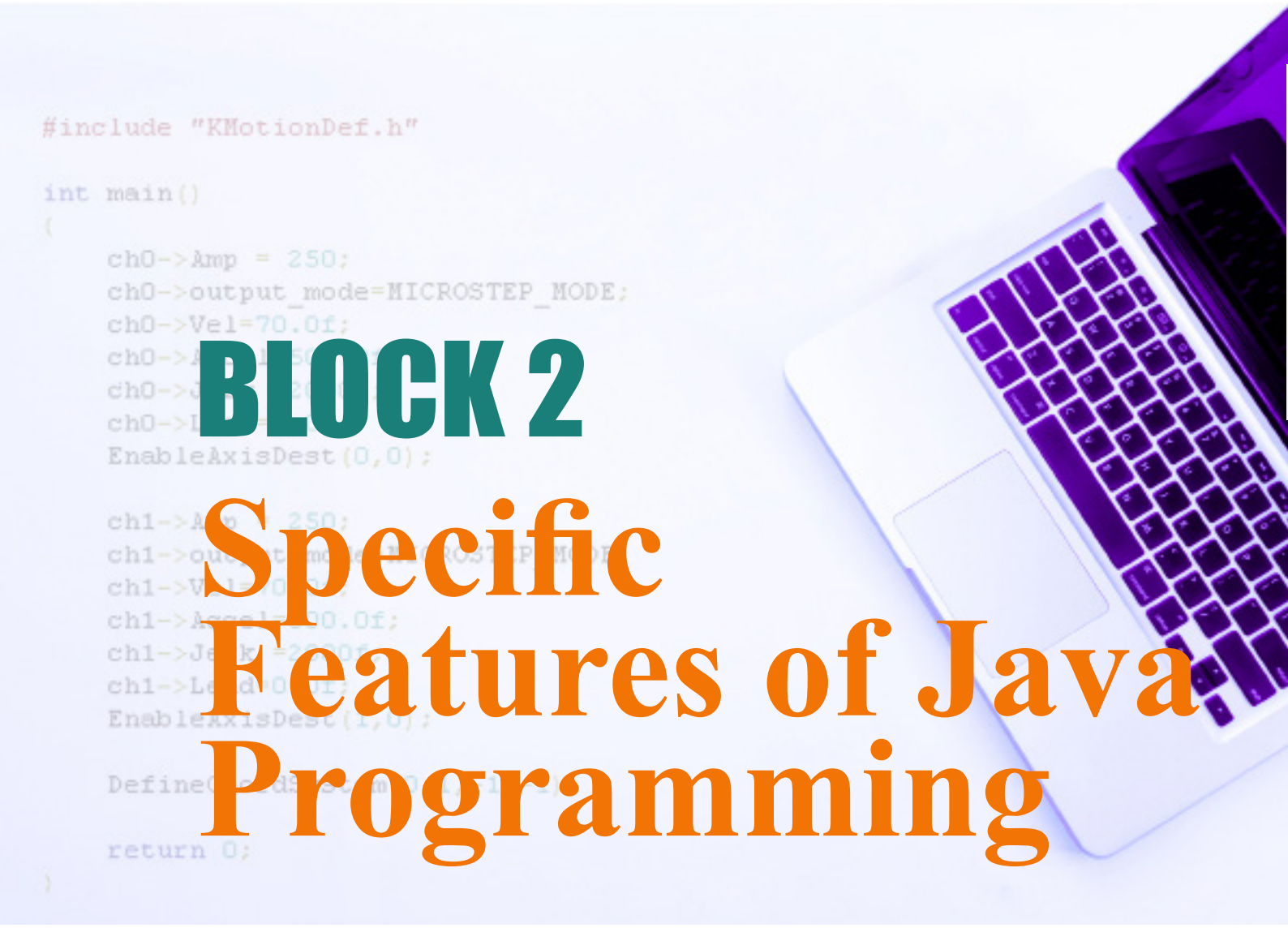
5. "Core Java Volume I – Fundamentals" by Cay S. Horstmann, 12th Edition, 2022, Pearson

Suggested Reading

1. Herbert, Schildt. "Java: The complete Reference 9th edition." (2014).

2. Balagurusamy, Emir. Programming in Java: A Primer. McGraw-Hill Education, 2010.

3. Sierra, Kathy, and Bert Bates. Head First Java: A Brain-Friendly Guide. " O'Reilly Media, Inc.", 2005.



BLOCK 2

Specific Features of Java Programming

String and String Buffer Class, Exception Handling

Learning Outcomes

At the end of this unit, the learner will be able to:

- identify the differences between a String and a StringBuffer in Java.
- familiarise the syntax for creating a String and a StringBuffer.
- list common methods available for manipulating Strings in Java.
- explain the purpose of exception handling in Java.
- explain the keywords used in exception handling, such as try, catch, and finally.

Prerequisites

You're probably familiar with storing numbers like integers or floating-point values in variables. Similarly, in programming, we use strings to store text, such as words or sentences. In Java, strings are more than just a sequence of characters - they are objects with special properties and methods for manipulating text.

Now, think about times when you've had to change a number, such as adding to a total or adjusting a value. In programming, we often need to modify text, or strings, in a similar way. For example, imagine you are creating a message that starts with "Hello" and later you want to add "World!" to form "Hello, World!" One way to do this would be to create a new string every time you make a change. However, this can be inefficient, especially if you're working with large amounts of text or frequently changing strings. This is where the **StringBuffer** class becomes useful. Instead of creating a new string every time you modify the text, StringBuffer allows you to make changes to the existing string more efficiently, like appending or inserting new characters without needing to create new objects each time. This not only saves memory but also makes the program run faster when performing multiple string modifications.

Think about a case that, when you made an error in basic math, like trying to divide a number by zero. You likely encountered an issue because dividing by zero is undefined, leading to a problem. Similarly, in Java, certain operations can lead to unexpected errors in your code. For instance, imagine a program where the user enters two numbers

for division. If the second number happens to be zero, attempting to divide by zero will cause the program to fail. This is where Exception Handling comes into play. In Java, exception handling is a method used to manage such errors, known as "exceptions," without causing the program to crash. For example, you can use a try-catch block to "catch" the error when it occurs and handle it gracefully, like displaying a message to the user instead of letting the program stop abruptly. By using exception handling, you ensure that your program can respond to unexpected situations, making it more robust and reliable.

Key Concepts

String Buffer, divide by zero, Java, exception handling, try, catch, finally

Discussion

2.1.1 String

In java programming language a string is an object that represents a sequence of characters.

2.1.2 Creation of string

2.1.2.1 By string literal

In Java, a String literal is defined by placing a sequence of characters within double quotes (“ “). When a string is written this way, Java treats it as an instance of the **String** class. For example:

String a= “java”;

2.1.3 String Constructor

2.1.3.1 By new keyword

In java, the **new** keyword can be used to create a new instance of a String object explicitly.

String a= new String(“java”);

The above code will creates a string java.

Similarly a char array is used to create a string object. To create a String initialized by an array of characters, use the constructor shown here:

String(char chars[ ])

For example:

```
Char ch[]={‘j’, ‘a’, ‘v’, ‘a’};
String s = new String(ch);
```

This constructor initializes s with the string "java".

A sample java program using above different methods of string creation is shown below:

```
public class StringExample{
        public static void main(String args[ ])
        {
                String s1="java";
                char ch[]={‘h’, ‘e’, ‘l’, ‘l’, ‘o’};
                String s2=new String(ch);
                String s3=new String("example");
                System.out.println(s1);
                System.out.println(s2);
                System.out.println(s3);
        }
}
```

The output from this program is as follows:

java

hello

example

2.1.4 String Length

The length of a string is the number of characters that it contains. To obtain this value, call the length( ) method, shown here:

```
int length( )
```

Example

```
char ch[] = { ‘j’, ‘a’, ‘v’, ‘a’ };

String s = new String(ch);

System.out.println(s.length());
```

Output:

4

2.1.5 String Concatenation

Typically, Java does not permit the use of operators on String objects, with the exception of the + operator. This operator is used to concatenate two strings, resulting in a new String object. It also allows multiple + operations to be chained together. For instance, the following example combines three strings into one.

```
String age = "10";

String s = "She is " + age + " years old.";

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

Output:

She is 10 years old.

2.1.6 Character Extraction

The String class offers several methods to extract characters from a String object. Some of these methods are discussed here. While the characters in a string cannot be accessed directly like elements in a character array, many String methods use an index or position to operate on the string. Similar to arrays, string indexing starts at zero.

2.1.6.1 charAt()

In Java, the charAt() method is utilized to obtain a character at a particular position within a String. The index denotes the character's position, with the first character being at index 0.

Syntax:

```
char character = stringvariable.charAt(index);
```

Example 1:

```
String str = “java”;

char ch =str.charAt(2);

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

Output:

v

Example 2:

```
String s = "Java Programming";
char ch = str.charAt(5);
System.out.println(ch);
```

Output:

P

2.1.6.2 getChars()

This method is used to extract multiple characters simultaneously from a string, Its general format is as follows:

void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart

The sourceStart indicates the starting index of the substring, while sourceEnd refers to the index just beyond the end of the desired substring. As a result, the substring includes characters from sourceStart up to sourceEnd - 1. The destination array, where the characters will be stored, is specified by target. The starting position within the target array, where the substring will be copied, is defined by targetStart. It is important to ensure that the target array has enough space to accommodate the number of characters in the selected substring.

The program below illustrates the use of the getChars() method:

```
class getCharsDemo {
public static void main(String args[]) {
String s = "This is a demo of the getChars method.";
int start = 10;
int end = 14;
char buf[] = new char[end - start];
s.getChars(start, end, buf, 0);
System.out.println(buf); } }
```

Output:

demo

2.1.6.3 getBytes( )

An alternative to getChars() is the getBytes() method, which stores characters in a byte array. This method utilizes the platform's default character-to-byte conversion. The simplest form of this method is as follows:

```
byte[ ] getBytes( );
```

2.1.6.4 toCharArray( )

To convert all the characters of a String object into a character array, the simplest approach is to use the toCharArray() method. This method returns a character array containing all the characters from the string. Its general syntax is as follows:

```
char[ ] toCharArray( )
```

Example:

```
String str = "Hello";
char[] charArray = str.toCharArray();
System.out.println(charArray);
```

Output:

Hello

2.1.7 String Comparison

The String class offers several methods for comparing strings or substrings within strings. Some of the important string comparison methods are listed below:

2.1.7.1 equals( )

The equals() method compares the content of two strings to determine if they are identical.

Syntax:

```
boolean equals(Object anotherObject);
```

Example:

```
String str1 = "Hello";
String str2 = "Hello";
String str3 = "World";
boolean result1 = str1.equals(str2);
boolean result2 = str1.equals(str3);
```

Output:

True

False

2.1.7.2 equalsIgnoreCase()

The equalsIgnoreCase() method compares two strings while disregarding any differences in case. This means it treats uppercase and lowercase letters as equivalent during the comparison.

Syntax:

int compareToIgnoreCase(String anotherString)

Example:

```
String str1 = "apple";
String str2 = "APPLE";
int result = str1.compareToIgnoreCase(str2);
```

Output:

True

2.1.7.3 compareTo()

The compareTo() method in Java compares two strings based on their Unicode values, determining their order in a lexicographical manner. This method, which belongs to the String class, returns an integer that shows how the strings relate to each other in terms of ordering.

Syntax:

int compareTo(String anotherString)

Example:

```
String str1 = "apple";
String str2 = "banana";
String str3 = "apple";
int result1 = str1.compareTo(str2);
int result2 = str1.compareTo(str3);
int result3 = str2.compareTo(str1);
System.out.println(result1);
System.out.println(result2);
System.out.println(result3);
```

Output:

-1

0

1

Lexicographic Order: This method compares strings by examining the Unicode value of each character. For instance, since the Unicode value of 'a' is 97 and 'b' is 98, the string "apple" is deemed less than "banana" because 'a' (97) precedes 'b' (98) in the Unicode sequence.

2.1.7.4 startsWith()

The startsWith() method is used to determine if a string starts with a specified string as prefix.

Syntax:

```
boolean startsWith(String prefix);
```

```
String str = "Hello World";
boolean startsWithHello = str.startsWith("Hello");
System.out.println("Does the string start with 'Hello'? " + startsWithHello);
boolean startsWithWorld = str.startsWith("World");
System.out.println("Does the string start with 'World'? " + startsWithWorld);
```

Output:

Does the string start with 'Hello'? true

Does the string start with 'World'? false

2.1.7.5 endsWith()

The endsWith() method is used to determine if a string ends with a specified string as suffix.

Syntax:

```
boolean endsWith(String suffix)
```

Example:

```
String str = "Hello World";
boolean endsWithHello = str.endsWith("Hello");
System.out.println("Does the string ends with 'Hello'? " + endsWithHello);
boolean endsWithWorld = str.endsWith("World");
System.out.println("Does the string ends with 'World'? " + endsWithWorld);
```

Output:

Does the string start with 'Hello'? False

Does the string start with 'World'? True

2.1.8 Searching Strings

The String class provides two methods that allow you to search a string for a specified character or substring:

- indexOf( )
- lastIndexOf( )

2.1.8.1 indexOf()

The indexOf() method in Java helps locate the position of a specific character or substring within a string. It returns the index of the first occurrence of the specified character or substring, or -1 if it isn't found.

Method 1: Syntax:

```
int indexOf(int ch)
```

Example

```
String str = "Hello World";
int index = str.indexOf("World");
System.out.println("Index of 'World': " + index);
```

Output:

6

Method2: indexOf(int ch, int fromIndex)

Finds the first occurrence of a specified character, starting the search from a specified index.

Syntax:

int indexOf(int ch, int fromIndex)

Example:

```
String str = "Hello World";
int index = str.indexOf('o', 5);
System.out.println("Index of 'o' starting from index 5: " + index);
```

Output:

7

2.1.8.2 lastIndexOf( )

The lastIndexOf() method in Java identifies the position of the final occurrence of a specific character or substring within a string. It functions like the indexOf() method but searches from the end of the string, returning the index of the last match. If the character or substring isn't found, it returns -1.

- lastIndexOf(int ch): Finds the last occurrence of a specified character.

Syntax:

int lastIndexOf(int ch);

Example:

```
String str = "Hello World";
int index = str.lastIndexOf('o');
System.out.println("Last index of 'o': " + index);
```

Output:

7

- lastIndexOf(String str): Finds the last occurrence of a specified substring.

Syntax:

int lastIndexOf(String str)

Example:

```
String str = "Hello World, Hello";
int index = str.lastIndexOf("Hello");
System.out.println("Last index of 'Hello': " + index);
```

Output:

13

- lastIndexOf(int ch, int fromIndex): Finds the last occurrence of a specified character, starting the search from a given index and searching backwards.

Syntax

```
int lastIndexOf(int ch, int fromIndex);
```

Example:

```
String str = "Hello World";
int index = str.lastIndexOf('o', 6);
System.out.println("Last index of 'o' before index 6: " + index);
```

Output

4

2.1.9 Changing the Case of Characters Within a String

There are two methods used to change case of characters within a string as follows:

- toUpperCase( )
- toLowerCase( )

2.1.9.1 toUpperCase()

This method is used to converts all characters in a string to uppercase.

Example:

```
String str = "hello";
String upperStr = str.toUpperCase();
System.out.println(upperStr);
```

Output

HELLO

2.1.9.2 toLowerCase()

This method is used to converts all characters in a string to lowercase.

Example:

```
String str = "HELLO";
String lowerStr = str.toLowerCase();
System.out.println(lowerStr);
```

Output:

hello

2.1.10 StringBuffer

In Java, StringBuffer represents a modifiable sequence of characters, that is similar to a String, but can be modified. As you know, String represents immutable character sequences with a fixed length. In contrast, StringBuffer represents a modifiable and expandable sequence of characters. It allows characters and substrings to be added either in the middle or at the end.

2.1.10.1 StringBuffer Constructors

StringBuffer offers these four constructors:

- StringBuffer( ) :The default constructor, which takes no parameters, allocates space for 16 characters without needing to resize.
- StringBuffer(int size) : It accepts an integer argument that explicitly sets the size of the buffer.
- StringBuffer(String str) :accepts a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.
- StringBuffer(CharSequence chars): Creates an object that contains the character sequence contained in chars and reserves room for 16 more characters.

2.1.11 Methods of StringBuffer

2.1.11.1 length( )

The length() method in StringBuffer provides the number of characters currently held in the buffer.

Example:

```
public class StringBufferLengthExample {
public static void main(String[] args) {
StringBuffer sb = new StringBuffer("Hello");
int length = sb.length(); // Print the length
System.out.println("Length: " + length);
}
}
```

Output:

5

2.1.11.2 capacity( )

This method is used to calculate the total allocated capacity.

```
public class StringBufferCapacityExample {
  public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("Hello");
    int capacity = sb.capacity();
    System.out.println("Capacity: " + capacity);
  }
}
```

Output:

21

Since sb is initialized with the string "Hello" when it is created, its length is 5. Its capacity is 21 because room for 16 additional characters is automatically added.

2.1.11.3 ensureCapacity( )

To allocate space for a specific number of characters in a StringBuffer after it has been initialized, you can use the ensureCapacity(int minimumCapacity) method. Here, minimumCapacity specifies the minimum size of the buffer.

Example

```
public class StringBufferEnsureCapacityExample {
  public static void main(String[] args) {
    StringBuffer sb = new StringBuffer();
    sb.ensureCapacity(50);
    int capacity = sb.capacity();
    System.out.println("Capacity after ensuring: " + capacity);
}
}
```

Output:

50

2.1.11.4 setLength( )

To adjust the length of the string in a StringBuffer object, you can use the setLength() method. The general syntax is as follows:

```
void setLength(int len);
```

Here, len defines the length of the string, and it must be a non-negative value.

2.1.11.5 charAt( )

The charAt(int index) method in StringBuffer retrieves the character at the given index. The indexing starts at 0, meaning the first character is at index 0, the second at index 1, and so on.

Syntax:

```
char charAt(int where);
```

Example:

```
public class StringBufferCharAtExample {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello World");
        char ch = sb.charAt(6);
        System.out.println("Character at index 6: " + ch);
  }
}
```

Output:

W

2.1.11.6 setCharAt( )

The setCharAt(int index, char ch) method in StringBuffer is used to change the character at a particular index. It replaces the character at the specified position with the new character you supply.

Syntax:

```
void setCharAt(int where, char ch);
```

Example:

```
public class StringBufferSetCharAtExample {
    public static void main(String[] args) {
            StringBuffer sb = new StringBuffer("Hello World");
            sb.setCharAt(6, 'J');
            System.out.println(sb);
    }
}
```

Output:

Hello World

2.1.11.7 getChars( )

The getChars() method is used to copy a substring of a StringBuffer into an array.

Syntax:

void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin);

- srcBegin: The starting index in the StringBuffer (inclusive).
- srcEnd: The ending index in the StringBuffer (exclusive).
- dst: The destination array to copy the characters into.
- dstBegin: The starting index in the destination array where copying will begin.

```
public class StringBufferGetCharsExample {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello World");
        char[] destArray = new char[5];
        sb.getChars(6, 11, destArray, 0);
        System.out.println(destArray);
}
}
```

Output:

World

2.1.11.8 append( )

The append() method in StringBuffer is used to add data to the end of the current buffer. It can append different types of data, including strings, characters, integers, and other objects, to the StringBuffer.

Syntax:

StringBuffer append(String str);

Example:

```
public class StringBufferAppendExample {
   public static void main(String[] args) {
       StringBuffer sb = new StringBuffer("Hello");
       sb.append(" World");
       sb.append(2024);
      System.out.println(sb);
 }
 }
```

Output:

Hello World2024

2.1.12 Exception Handling

An exception is an unexpected condition that occurs during the execution of a program. It happens while the program is running, disrupting its normal flow. In simple terms, an exception represents a runtime error in the code.

2.1.12.1 Exception-Handling Fundamentals

Java handles exceptions using five key terms: **try**, **catch**, **throw**, **throws**, and **finally**. These keywords work together to manage and handle errors during program execution.

- try: The program statements you want to check for exceptions are placed inside a try block.
- catch: In Java, the catch block is responsible for handling exceptions that might arise in the corresponding try block. When an exception is thrown within the try block, the catch block intercepts it, allowing the program to manage the error without terminating
- throw: In Java, the throw keyword is used to manually trigger an exception, signaling an error or special condition in a method. This exception can be handled in the current method or passed to another part of the program for handling.
- throws: Any exception that is thrown out of a method must be specified as such by a throws clause.
- finally: Any code that absolutely must be executed after a try block completes is put in a finally block.

General form of an exception-handling block:

```
try {
      // block of code to monitor for errors }
}
catch (ExceptionType1 exOb) {
  // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
 }
finally {
// block of code to be executed after try block ends
 }
```

Here, ExceptionType is the type of exception that has occurred.

2.1.13 Exception Types

In Java, all exception types derive from the built-in class *Throwable*, which sits at the top of the exception hierarchy. *Throwable* is divided into two main subclasses: *Exception* and *Error*, each handling different categories of exceptions. *Exception* is intended for conditions that programs should handle, including its subclass *RuntimeException*, which covers errors like division by zero or out-of-bounds array access. Conversely, *Error* deals with exceptions that are generally not intended to be caught by programs are shown in the figure 2.1.1

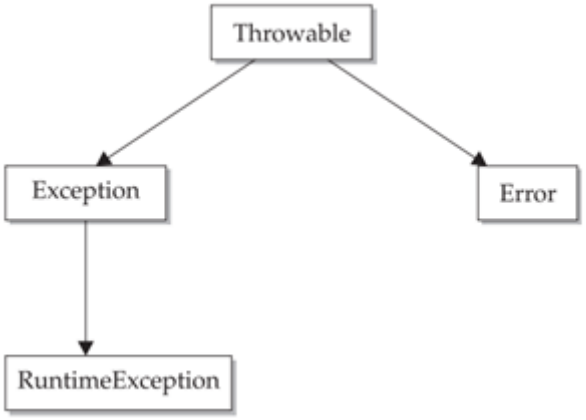


Figure 2.1.1 Types of Exception

2.1.14 Understanding Unhandled Exceptions in Java

Before learning how to handle exceptions, it's important to understand what happens when they are not managed. The following program intentionally triggers a divide-by-zero error:

```
class Exc0 {
 public static void main(String args[]) {
   int d = 0;
   int a = 42 / d;
 }
}
```

When the Java runtime detects this error, it creates and throws a new exception object, stopping the program's execution. Since there is no custom exception handler, the default handler catches the exception, shows an error message, prints a stack trace, and ends the program. In this case, the exception thrown is an ArithmeticException, a specific subclass of Exception, indicating the division by zero error.

2.1.14.1 Exception handling using try and catch

Consider the following program example:

```
class Exc2 {
public static void main(String[] args) {
 int d, a;
 try { // This block is being monitored for exceptions
        d = 0;
        a = 42 / d; // This will cause a divide-by-zero error
        System.out.println("This will not be printed.");
}
catch (ArithmeticException e) { // This block catches the ArithmeticException caused by division by zero
System.out.println("Division by zero."); } // The program continues after the catch block System.out.println("After catch statement.");
}
}
```

try block: The try block is used to observe code that might throw an exception. In the given program, the code inside the try block tries to execute the operation a = 42 / d, where d has been assigned a value of zero. This operation results in a division-by-zero error, which is considered an exceptional condition in Java. By enclosing this code in the try block, the program can intercept the error before it leads to a crash.

Exception Triggered: When the division-by-zero operation takes place, the Java run-time system recognizes it as an ArithmeticException. Rather than abruptly terminating the program, Java generates this exception. The raised exception serves as an indication that an error has occurred, and it must be addressed to enable the program to recover from this error in a smooth manner.

catch block: The catch block directly follows the try block and is intended to manage specific types of exceptions. In this scenario, it addresses the ArithmeticException that arises from attempting division by zero. When the exception is triggered, the control flow shifts from the try block to the catch block, where the error is handled. Within the catch block, a straightforward message, "Division by zero," is displayed to notify the user of the error. This mechanism prevents the program from crashing and enables it to continue running after the error has been addressed.

Program Continuation:

Once the exception is addressed within the catch block, the program continues its normal execution with the statement that follows the entire try-catch construct. In this instance, the program outputs "After catch statement." This illustrates that by managing the

exception, the program avoids an early termination and can proceed to run seamlessly.

2.1.14.2 throw

The throw keyword is used to manually generate an exception in Java. It signals an exceptional condition during program execution. A throw statement is typically followed by an exception object that specifies the type and details of the exception being raised.

How it applies to the example:

In the given example, the division-by-zero operation (a = 42 / d) automatically triggers an ArithmeticException, but the throw keyword could be explicitly used to raise the same exception.

Modified Code with throw:

```
class Exc2 {
    public static void main(String[] args) {
        try {
            int result = divide(42, 0);
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
        System.out.println("After catch statement.");
    }
    static int divide(int numerator, int denominator) throws ArithmeticException {
      if (denominator == 0) {
          throw new ArithmeticException("Cannot divide by zero.");
      }
      return numerator / denominator;
    }
}
```

The divide method declares that it may throw an ArithmeticException using the throws keyword. When the exception occurs, it is passed to the caller (the main method), which then handles it in the try-catch block.

2.1.14.3 throws:

The throws keyword is used in a method declaration to indicate that the method might throw one or more exceptions. It informs the caller of the method that they need to handle these exceptions.

If the division operation was part of a separate method, the throws keyword would be used to declare the potential exception.

Modified Code with throws:

```
class Exc2 {
    public static void main(String[] args) {
        try {
            int result = divide(42, 0);
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
        System.out.println("After catch statement.");
    }

    // Method that throws ArithmeticException
    static int divide(int numerator, int denominator) throws ArithmeticException {
        if (denominator == 0) {
            throw new ArithmeticException("Cannot divide by zero.");
        }
        return numerator / denominator;
    }
}
```

The divide method declares that it may throw an ArithmeticException using the throws keyword. When the exception occurs, it is passed to the caller (the main method), which then handles it in the try-catch block.

2.1.14.4 finally

The finally block is always executed after the try block, regardless of whether an exception was thrown or caught. It is typically used for cleanup actions such as closing files, releasing resources, or resetting variables.

A finally block can be added to the example to ensure that a specific piece of code runs no matter what happens during exception handling.

Modified Code with finally:

```
class Exc2 {
    public static void main(String[] args) {
        int d, a;
        try {
            d = 0;
            a = 42 / d; // This will cause a divide-by-zero error
            System.out.println("This will not be printed.");
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: Division by zero.");
        } finally {
            System.out.println("Finally block executed.");
        }
        System.out.println("After try-catch-finally.");
    }
}
```

The finally block ensures that the message "Finally block executed." is always printed, even if an exception occurs or does not occur.

It demonstrates that resource cleanup or mandatory actions can take place, regardless of exception handling.

Recap

Strings:

- Strings represent text in Java and are immutable (cannot be changed once created).
- Common methods for string manipulation include:
  - **length():** Returns the length of the string.
  - **charAt(index):** Returns the character at a specific index.
  - **substring(start, end):** Extracts a portion of the string.
  - **indexOf(char):** Finds the position of a character or substring.
  - **toUpperCase()/toLowerCase():** Converts all characters to upper or lower case.
  - **concat(String str):** Concatenates two strings.

StringBuffer:

1. StringBuffer allows efficient string modification without creating new objects.
2. Common methods of StringBuffer include:
   - **append(String str):** Adds a string to the end of the buffer.
   - **insert(offset, String str):** Inserts a string at a specified position.
   - **delete(start, end):** Removes characters within a specified range.
   - **reverse():** Reverses the entire string.
   - **replace(start, end, String str):** Replaces characters between specified positions.
   - **setLength(int newLength):** Sets the length of the buffer.

Exception Handling:

1. Exceptions are runtime errors, like dividing by zero or accessing an invalid index.
2. Java's Exception Handling uses try-catch blocks to handle errors.
3. Methods like getMessage() help retrieve error details.
4. Ensures the program continues to run smoothly by catching and resolving errors during execution.

Objective Type Questions

1. What is the method used to find the length of a string?
2. Which method retrieves a character at a specific index in a string?
3. What method extracts a portion of a string?
4. Which method finds the position of a character in a string?
5. What is the method to convert all characters in a string to uppercase?
6. Which class allows efficient modification of strings?
7. What method adds a string to the end of a StringBuffer?
8. Which method inserts a string at a specified position in a StringBuffer?
9. What is the technique used to handle runtime errors in Java?
10. Which keyword is used to start a block for handling exceptions?
11. What keyword is used to define a block for handling caught exceptions?
12. What type of error is caused by dividing by zero?

Answers to Objective Type Questions

1. length()
2. charAt
3. substring
4. indexOf
5. toUpperCase
6. StringBuffer
7. append
8. Insert
9. Exception

10. Try

11. Catch

12. ArithmeticException

Assignments

1. What is a string in Java, and why are strings considered immutable? Illustrate your answer with a code example that shows string creation and its immutability.

2. Identify and explain three methods from the String class in Java. Include code snippets that demonstrate how each method functions.

3. Describe the purpose of StringBuffer in Java. How does it allow for more efficient string manipulation compared to the String class? Provide an example that shows the use of the insert method.

4. What does exception handling achieve in Java? Discuss the structure and purpose of try-catch blocks, and provide an example that highlights their effectiveness in managing errors.

5. Give an example of a potential runtime error in Java code. How would you implement exception handling to manage this error, and what methods would you use to retrieve error information?

References

1. Bates, Bert, and Kathy Sierra. *Head First Java*. 2nd ed., O'Reilly Media, 2005.

2. Rajshekhar, Sharanam Shah, and Vaishali Shah. *JDBC, Servlets, and JSP Black Book*. Dreamtech Press, 2011.

3. Evans, David R., and John C. Debs. *Database Programming with JDBC and Java*. 2nd ed., O'Reilly Media, 2000.

4. Eckel, Bruce. *Thinking in Java*. 4th ed., Prentice Hall, 2006.

5. Zakhour, Sowmya, et al. *The Java Tutorial: A Short Course on the Basics*. 6th ed., Addison-Wesley, 2015.

Suggested Reading

1. Herbert Scheldt, Java The Complete Reference, 8th Edition, Tata McGraw-Hill Edition, ISBN: 9781259002465

2. E Balaguruswamy, Programming in Java: A Primer, 4th Edition, Tata Mcgraw Hill Education Private Limited, ISBN: 007014169X.

3. Kathy Sierra, Head First Java, 2nd Edition, Shroff Publishers and Distributors Pvt Ltd, ISBN: 8173666024.

Multithreading

Learning Outcomes

The students will be able:

- familiarise multithreading concept in Java
- understand creation of a thread using Thread class
- learn to create a thread using Runnable interface

Prerequisites

Imagine you're in a busy restaurant where multiple chefs are preparing different dishes simultaneously. One chef is frying vegetables, another is boiling pasta, and yet another is grilling meat. Each chef works independently, focusing on their specific task, but together they create a delicious meal for the customers. This seamless coordination allows the restaurant to serve food quickly, enhancing the dining experience.

Now, think about how this scenario relates to computers. Just like the chefs, a computer can handle multiple tasks at once. When you listen to music while browsing the internet or download files while typing a document, your computer is multitasking. In computer science, this concept is known as **multithreading**, where several tasks (or threads) run concurrently, improving efficiency and speed.

Before diving into multithreading, you should be familiar with basic programming concepts, such as control structures (like loops and conditionals) and object-oriented programming principles, including classes and methods. Understanding these fundamentals will help you grasp how threads function and interact within a program. Just like a well-coordinated kitchen, where each chef knows their role, mastering these basic concepts will prepare you for the complexities of multithreading in programming.

Keywords

Multitasking, Multithreading, Thread, Thread class, Runnable interface

Discussion

2.2.1 MultiTasking

Suppose your teacher is teaching multithreading in the class. Some students are very studious and they are listening the class so seriously. Some students are writing lecture notes. Some are just looking at the teacher's face and thinking about something else. While some others are watching the environment. Here, several activities are going on at the same time, which is called multitasking. Similarly in Computer Science, executing several tasks simultaneously is called multitasking. For example, when you are typing a Java program in the editor you can play an audio in the background. And also you can download a file from the internet at the same time. This is called multitasking.

Multitasking is broadly classified into two - Process based multitasking and Thread based multitasking.

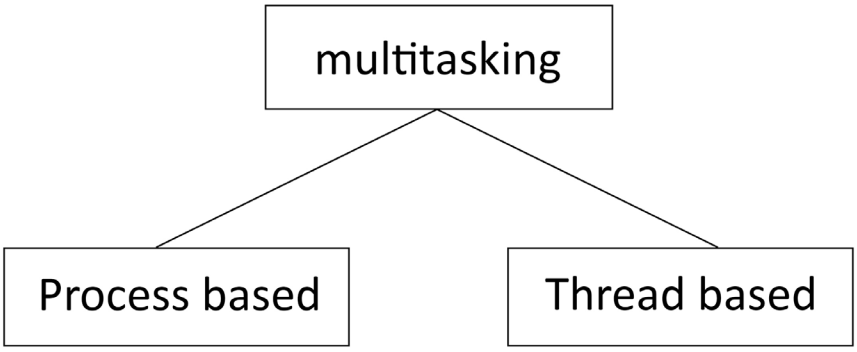


Fig 2.2.1 Classification of multitasking

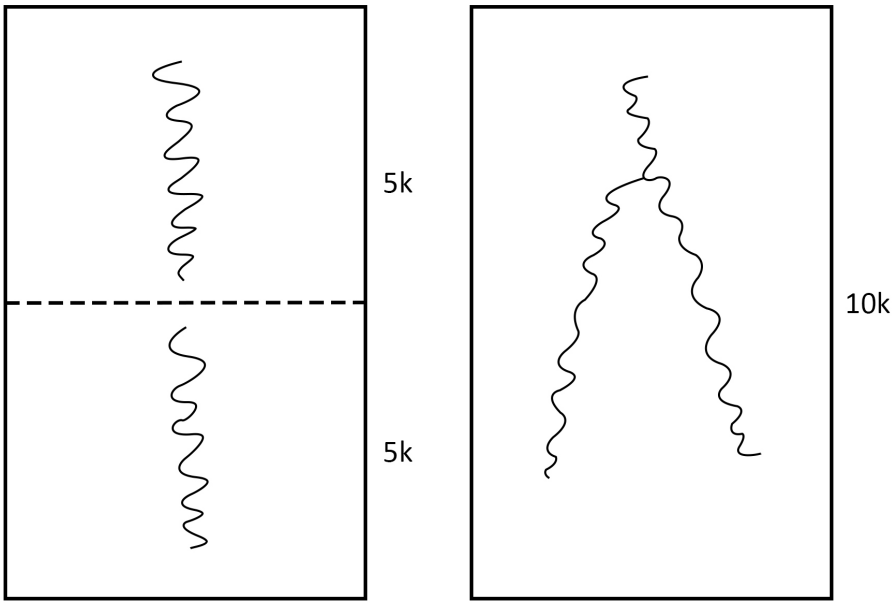
1. Process based multitasking: In the above example of writing Java program, playing music and downloading file from the internet, each task is a separate independent process. There is no dependency between each other. This is called process based mutitasking. i e, Executing several tasks simultaneously where each task is a separate independent process is called process based multitasking.

2. Thread based multitasking: Suppose a programmer has written a Java program which contains 10K lines of code. In the normal program execution the codes are executed sequentially.i e , line by line. But the programmer finds a fact that the first 5K lines of code is independent of the second 5K lines of code. Then why does he have to wait for the completion of the first 5K lines of code to execute the remaining 5K lines of code. Isn't possible to execute both parts simultaneously? The answer is Yes. It is possible with the assistance of Multithreading.

Executing several tasks simultaneously where each task is the separate independent parts of the same program is called ***Multithreading.***

Simultaneous execution of different parts of the same program will reduce the execution time significantly.

Multithreading has various applications across different domains. It is used in web servers to handle multiple client requests simultaneously, improving response times and resource utilization. In user interfaces, it helps keep the interface responsive while performing background tasks, such as loading data or processing.



(a) **(b)**

Fig. 2.2.2 (a) Program execution without multitasking (b) Program execution with multitasking

In game development, multithreading manages graphics rendering, input handling, and physics calculations in parallel to enhance performance and user experience. It speeds up data analysis and processing tasks by dividing workloads across multiple threads, especially in big data applications. Scientific simulations run complex calculations in parallel to reduce computation time. It ensures timely responses to events in real-time systems like robotics and embedded systems. In multimedia applications, it processes audio, video, and graphics in parallel to improve playback quality and reduce latency. In cloud computing, it enhances resource management and task execution by distributing workloads across multiple threads. Additionally, in machine learning, multithreading trains models more efficiently by parallelizing computations across multiple threads, especially for large datasets.

2.2.2 Defining a thread

We can define a thread in the following two ways

- By extending Thread class

- By implementing Runnable interface

2.2.2.1 By Extending Thread class

There is a class in Java named the **Thread** class that provides constructors and methods to create and perform operations on a thread. We can extend this class for defining a thread. By definition a thread is a flow of execution. Each thread has a job to perform. To create a thread by extending Thread class, follow the given steps:

1. Define a thread

Suppose you want to write a Java program to print a statement 10 times using thread. In the first step create a user defined class by extending the Thread class. See the following code.

```
class MyThread extends Thread
{
        public void run( )
        {
                for (int i = 0; i < 10; i++)
                {
                        System.out.println("Child thread");
                }
         }
  }
```

This is called defining a thread. The job of the thread is written inside the run( ) method. There is a run( ) method in the Thread class. Here we are overriding that run( ) method.

2. Instantiation of thread and start the thread

Every Java program starts from the main class. So the next step is to write the main method. Inside the main method an object of the user defined thread class (here it is MyThread) is created and the thread is started with the start( ) method. The code is given below.

```
class DemoThread
{
        public static void main(String[] args)
        {
                MyThread t = new MyThread( );       // Thread instantiation
```

```
            t.start( );                                    // Starting of thread
            for (int i = 0; i < 10; i++)
            {
                    System.out.println("Main thread");
            }
        }
}
```

The program can be written as:

```
class MyThread extends Thread
{
  public void run( )
    {
       for (int i = 0; i < 10; i++) {
         System.out.println("Child thread");
       }
     }
class DemoThread
{
    public static void main(String[ ] args) {
        MyThread t = new MyThread( );        // Thread instantiation
        t.start( );  // Starting of thread
        for (int i = 0; i < 10; i++) {
            System.out.println("Main thread");
        }
    }
}
```

There is only one thread up to the statement MyThread t = new MyThread( ) which is the main thread. From the statement t.start( ) onwards there are two threads – one executed by the main thread and the other by the child thread. The start( ) method is

responsible for creating a new flow of execution. Instead of t.start( ) if you are writing t.run( ) (i e , calling the run( ) method directly), no separate thread will be created. Since more than one thread runs simultaneously there is no guaranteed output for these types of programs. Which thread is to be executed at a particular time is decided by a program called *Thread scheduler* which is a part of the JVM. The output may vary from run to run or system to system. Some possible outputs are shown below.

Output 1:

Main thread

Main thread

Main thread

Main thread

Main thread

Main thread

Main thread

Main thread

Main thread

Main thread

Child thread

Child thread

Child thread

Child thread

Child thread

Child thread

Child thread

Child thread

Child thread

Child thread

Output 2:

Child thread

Child thread

Main thread

Main thread

Child thread

Main thread

Child thread

Child thread

Main thread

Child thread

Main thread

Child thread

Main thread

Main thread

Child thread

Main thread

Main thread

Child thread

Child thread

Main thread

One important point you have to keep in mind is that we are using multithreading for independent jobs. If there is a dependency between the jobs, don't go for multithreading.

2.2.2.2 By implementing Runnable interface

In the first approach, we extended the Thread class. And the Thread class already implemented Runnable interface. Here, in the second approach we are going to implement the Runnable interface directly. The Runnable interface is present in the java.

```
class MyRunnable implements Runnable {
  public void run() {
    for (int i = 0; i < 10; i++) {
      System.out.println("Child thread");
    }
  }
```

```
}
class DemoThread {
  public static void main(String[ ] args) {
    MyRunnable runnable = new MyRunnable( );  // Runnable instantiation
    Thread t = new Thread(runnable);  // Passing Runnable to Thread
    t.start( );  // Starting the thread

    for (int i = 0; i < 10; i++) {
      System.out.println("Main thread");
    }
  }
}
```

lang package and it contains only one method – run( ) method. The steps for writing the above example program is given below.

1. Create a class that implements the Runnable interface and write the code for the run( ) method inside it.

2. Then, create a Thread object and pass your Runnable class to the Thread constructor, since Thread can accept any Runnable object.

3. To start the thread, call the start( ) method on the Thread object. This will create a new thread and run the code inside the run( ) method.

If you just call run( ) directly, the code will execute in the current thread, not a new one. To run it in a separate thread, you must call start( ).

Here also output can vary from run to run as in the first approach.

Recap

- **Multitasking**: Simultaneous execution of multiple tasks.
- **Types of Multitasking**:
  - **Process-based Multitasking**: Independent processes executing simultaneously.
  - **Thread-based Multitasking**: Multiple threads running independent parts of the same program.
- **Benefits of Multithreading**:
  - Reduces execution time.
  - Enhances responsiveness in user interfaces.
  - Improves performance in applications like web servers and game development.
- **Defining a Thread**:
  - **By extending Thread class**: Override the run() method to define thread behavior.
  - By implementing Runnable interface: Create a class that implements Runnable, and define the run( ) method.
- **Creating and Starting a Thread:**
  - Instantiate a thread object (e.g., MyThread t = new MyThread( )).
  - Start the thread using t.start() to create a new flow of execution.
- **Thread Scheduler**: Decides which thread runs at a given time; output may vary with each execution.

Objective Type Questions

1. What is the term for executing multiple tasks simultaneously in a computer system?
2. Which type of multitasking involves independent processes?
3. What is the method called that must be overridden to define a thread when extending the Thread class?
4. What is the interface implemented to create a thread without extending the Thread class?

5. What is the primary benefit of using multithreading in applications?
6. Which Java interface must a class implement to create a thread?
7. Which method is used to start a thread in Java?
8. What is the default priority of a thread in Java?
9. Which method is used to put a thread to sleep for a specified amount of time?
10. What happens if the run() method of a thread is called directly instead of calling start()?
11. Which keyword in Java is used to prevent thread interference in a critical section?
12. What is the state of a thread after the yield() method is called?
13. Which exception is thrown if the sleep() method is interrupted?
14. What happens when the join() method is called on a thread?

Answers to Objective Type Questions

1. Multitasking
2. Process
3. run
4. Runnable
5. Performance
6. Runnable
7. start()
8. 5 (Thread.NORM_PRIORITY)
9. sleep(milliseconds)
10. Executes like a normal method in the calling thread
11. synchronized
12. Runnable

13. InterruptedException.

14. The calling thread waits until the specified thread completes.

Assignments

1. Write a Java program that demonstrates multithreading by creating a scenario where multiple threads print messages concurrently. Each thread should print its message ten times, and the main thread should also print a different message ten times. Use both the Thread class and the Runnable interface in your implementation.

2. Write a Java program that demonstrates multithreading by creating two threads: one thread that prints the numbers from 1 to 10 and another thread that prints the alphabet letters from A to J. Use both approaches—extending the Thread class and implementing the Runnable interface—to accomplish this task.

References

1. Bates, Bert, and Kathy Sierra. *Head First Java*. 2nd ed., O'Reilly Media, 2005.

2. Rajshekhar, Sharanam Shah, and Vaishali Shah. *JDBC, Servlets, and JSP Black Book*. Dreamtech Press, 2011.

3. Evans, David R., and John C. Debs. *Database Programming with JDBC and Java*. 2nd ed., O'Reilly Media, 2000.

4. Eckel, Bruce. *Thinking in Java*. 4th ed., Prentice Hall, 2006.

5. Zakhour, Sowmya, et al. *The Java Tutorial: A Short Course on the Basics*. 6th ed., Addison-Wesley, 2015.

Suggested Reading

1. Java: The Complete Reference by Herbert Schildt

2. Java Concurrency in Practice by Brian Goetz

3. Effective Java by Joshua Bloch

4. Java Threads" by Scott Oaks

UNIT 3 Applets and Event Handling

Learning Outcomes

The learner will be able to:

- to describe what a Java applet is and how it differs from a standalone application.
- list the primary lifecycle methods of applets and describe their functions.
- define event handling and identify common user actions that trigger events in applets.
- to enumerate common event listener interfaces in Java.

Prerequisites

Imagine you're browsing the web in the early 2000s and come across a page that's more than just text and images. You notice a small interactive element, perhaps a game or a calculator, running directly inside the browser. No downloads, no installations - just seamless interactivity embedded in the web page itself. What you're seeing is called an applet, a simple Java program created to work inside a web browser. Applets were an early way to bring dynamic content to websites, offering interactivity and functionality long before modern web technologies like HTML5 or JavaScript became common. But what makes these applets react to your actions, like clicking buttons or moving your mouse? That's where event handling comes into play.

Before we dive into applets and event handling, it helps to understand some basic concepts of Java programming. Java is a versatile language capable of creating programs that interact with users in many ways. In the case of applets, which run directly within a web page, event handling is what makes them interactive. To understand applets and event handling better, you should first have a basic grasp of how Java programs work, particularly object-oriented principles and how graphical user interfaces (GUIs) process events.

Key Concepts

Applets, Event handling, Event source, Event object, Event listener, Delegation event model, Event listener interfaces and Classes

Discussion

2.3.1 Introduction to Applets and Event Handling

Applets in Java are small programs that can be embedded in web pages to make them interactive. Unlike regular applications, applets run inside a web browser or applet viewer, making them platform-independent. They extend the java.applet.Applet class and follow a lifecycle with methods like init(), start(), stop(), and destroy(), which manage their initialization and shutdown processes.

A key feature of applets is event handling, which makes them interactive by responding to user actions like clicks or key presses. Java's Abstract Window Toolkit (AWT) uses a system where events (like mouse clicks) are generated by sources (like buttons) and handled by listener objects that implement interfaces such as ActionListener or MouseListener. This enables applets to react to user inputs in real time.

By combining applets and event handling, developers can create dynamic web components that interact with users, making websites more engaging. The java.applet package supports this functionality, providing tools for applets to communicate with the web environment. Though applets are less common today, they were once essential for adding interactive elements to web pages.

2.3.1.1 Overview of Java Applets

Java applets are small, platform-independent applications that run within a web browser or applet viewer, allowing developers to add dynamic, interactive elements to web pages. They extend the java.applet.Applet class and follow a structured lifecycle with methods like init(), start(), stop(), and destroy(), managing the applet's initialization, execution, and termination phases. Applets were popular in the early web era for tasks like animations and small games, offering cross-platform functionality without compatibility issues.

A key feature of Java applets is their ability to handle user interactions through event handling. Using Java's event-delegation model, applets can respond to actions such as mouse clicks or key presses with the help of event listeners. Though applets have largely been replaced by modern web technologies like HTML5 and JavaScript, they were an important early example of embedding interactive, web-based applications in a cross-platform environment.

- **Applet Lifecycle Methods**

Java applets go through a series of steps, called lifecycle methods, which manage how they run from start to finish. These methods - init(), start(), stop(), and destroy() - are automatically triggered by the browser or applet viewer at different points in the applet's life. Each method plays an important role in ensuring that the applet functions properly while it's being used.

The first method, init(), is called when the applet is loaded. It sets up all the necessary components, like initializing variables or loading images, so the applet is ready to run.

After this, the start() method begins the applet's execution, such as starting an animation or reacting to user interactions. The applet continues running as long as the user stays on the web page.

When the user leaves the page, the stop() method is called to pause the applet. This means that ongoing activities like animations or timers are temporarily halted. If the user comes back, the start() method can resume the applet's actions. Finally, when the applet is no longer needed, the destroy() method is called to free up resources and ensure everything is cleaned up properly, like memory and other system resources.

These methods help manage an applet's performance and resource usage, making sure it runs efficiently and can adapt to changes in the user's actions or the webpage environment.

- **Creating and Running Applets**

Creating a Java applet begins by writing a Java class that extends the java.applet.Applet class. This class must include the necessary lifecycle methods like init(), start(), stop(), and destroy(), which control how the applet behaves at different stages. Developers can also add code to handle user interactions, such as responding to mouse clicks or key presses, making the applet interactive.

After the applet's code is written, it is compiled just like any other Java program using the Java Development Kit (JDK). However, running an applet is different from running a typical Java application. Instead of running it directly, applets are designed to run inside a web browser or an applet viewer. To make this happen, an HTML file is created, which contains special tags (like the <applet> tag) that specify the applet's class and any parameters it needs. Alternatively, developers can use the Java applet viewer tool to run the applet without needing a browser.

Although applets used to be popular for adding interactive features to web pages, most modern web browsers no longer support them due to security concerns and the rise of newer technologies like HTML5 and JavaScript. However, applets can still be run in specific environments like the applet viewer for testing or learning purposes, making them an interesting part of Java's early role in web development.

2.3.1.2 Introduction to Event Handling in Java

Event handling in Java is a crucial mechanism that allows applications to respond to user actions, such as mouse clicks, keyboard inputs, or other interactions. Java's event handling framework is designed to provide a seamless way for developers to create interactive applications by detecting and responding to events generated by user actions or system occurrences. This process enhances the user experience by allowing the application to react dynamically to inputs.

In Java, events are typically generated by components of the user interface (UI), such as buttons, text fields, and menus. To handle these events, Java employs an event-delegation model, where event sources, like a button, create an event when a user interacts with them. These events are then sent to listener objects that implement specific interfaces to handle them appropriately. Common interfaces include ActionListener,

MouseListener, and KeyListener, each designed to handle particular types of events. For example, ActionListener is used for button clicks, while MouseListener captures mouse movements and clicks.

Developers can register listeners to specific components to define how the application should respond to various events. This registration is done through methods like addActionListener() for buttons, which links a listener to the button so that it can respond when the button is clicked. This separation of event handling from the main application logic allows for cleaner and more organized code, making it easier to maintain and extend.

Understanding Events and Listeners

In Java, an event is an object that describes a change in the state of a source, like a button press or a window being closed. A listener is an object that "listens" for events and defines how to respond to those events when they occur. The key components of event handling include:

- Event Source: The component that generates the event, such as a button or text field.
- Event Object: An object that encapsulates information about the event, such as ActionEvent or MouseEvent.
- Event Listener: An interface that must be implemented by any class interested in handling a specific type of event. Some common listeners include ActionListener, MouseListener, and KeyListener.

Steps in Event Handling

To handle events in Java, developers follow a few steps:

1. Registering a Listener: An event source must register an event listener using methods like addActionListener() or addMouseListener().
2. Implementing Event Handling Methods: The listener must implement methods defined by the event listener interface, such as actionPerformed() for handling action events.
3. Generating Events: Once an event is generated (e.g., when a user clicks a button), the corresponding method in the registered listener is invoked, and the desired action is executed.

Event Sources and Event Objects

In Java's event-handling mechanism, event sources and event objects are key components of the delegation event model, allowing programs to respond to user actions and other events effectively.

Event Sources

An event source is the object that generates an event when its state changes. Any

interactive component in a Java GUI, such as a button, checkbox, or text field, can act as an event source. When a user interacts with these components, the event source triggers an event to signal that something has occurred. The event is then sent to any registered listeners that have expressed interest in handling that type of event.

Examples of event sources include:

- JButton (generates an ActionEvent when clicked)
- JTextField (generates a TextEvent when the text changes)
- JMenuItem (generates an ActionEvent when selected)

To make an event source meaningful, the source must register one or more listeners to respond when the event occurs. This is usually done through methods like addActionListener(), addMouseListener(), or addKeyListener(), depending on the type of event.

Event Object

An event object is an instance of a class that represents the details of the event that has occurred. It carries information about the event source, the type of event, and any additional data related to the event. All event objects in Java are derived from the java.util. EventObject class, which contains a reference to the event source.

There are different types of event objects, depending on the type of event being generated. Some common event objects include:

- ActionEvent: Represents events triggered by button clicks, menu selections, etc.
- MouseEvent: Represents mouse actions such as clicks, movements, and drags.
- KeyEvent: Represents keyboard actions, such as key presses and key releases.
- WindowEvent: Represents window-related actions, such as opening, closing, or minimizing a window.

Each event object contains methods to access information specific to the event. For instance, an ActionEvent includes methods like getActionCommand() to retrieve the command associated with the action, while a MouseEvent provides methods to retrieve the coordinates of a mouse click (getX(), getY()).

Example

Below is an example demonstrating the relationship between an event source and an event object:

```
import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;
```

```
import javax.swing.JButton;
import javax.swing.JFrame;
public class EventExample
{
	public static void main(String[] args)
	{
		JFrame frame = new JFrame("Event Example");
		JButton button = new JButton("Click Me");
				// Register an ActionListener to the button
		button.addActionListener(new ActionListener()
		{
			public void actionPerformed(ActionEvent e)
			{
				// The ActionEvent object provides details of the event
				System.out.println("Event Source: " + e.getSource());
				System.out.println("Action Command: " +
				e.getActionCommand());
			}
		});
	frame.add(button);
	frame.setSize(300, 200);
	frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	frame.setVisible(true);
	}
}
```

In this example:

- The event source is the button (JButton), which generates an ActionEvent.
- The event object (ActionEvent) contains details such as the source of the event (e.getSource()) and the action command (e.getActionCommand()).

2.3.2 Delegation Event Model

The Delegation Event Model is the foundation of how Java handles events in graphical user interfaces (GUIs). It's designed to make responding to user actions - like clicking a button, moving the mouse, or pressing a key - both efficient and easy to manage. The idea is simple: when something happens, instead of the object responsible for generating the event handling it directly, the task is "delegated" to another object that is specifically designed to deal with it.

2.3.2.1 Key Features

There are three key pieces to understand how the model works:

1. Event Source: This is the object where the event originates—think of a button, a text field, or a menu item. When a user interacts with one of these, it triggers an event. For example, clicking a button creates an action event.

2. Event Listener: A listener is an object that's interested in what happens to the event source. The listener "listens" for a specific event, such as a button being clicked, and responds when that event occurs. You register a listener with the event source, usually using methods like addActionListener() or addMouseListener().

3. Event Object: This object holds all the details about the event—what caused it, where it came from, and sometimes even information about the interaction, like where the mouse was clicked. The event object is passed to the listener when the event is triggered, giving the listener all the info it needs to handle the event.

2.3.2.2 How the Delegation Event Model Works

Here's a simple breakdown of how the delegation event model works in practice:

1. Generating the Event: When a user interacts with the event source (like clicking a button), the event source generates an event object to represent what just happened.

2. Delegating the Event: The event source doesn't handle the event itself. Instead, it passes this responsibility to the event listener, which has been registered with the event source.

3. Handling the Event: The event listener's method (like actionPerformed()) is called, and the listener takes action based on the details provided by the event object.

This process separates the "what happened" from the "what to do when it happens," making code cleaner and easier to maintain.

2.3.2.3 Importance of Delegation Event Model

1. Clear Organization: It separates the event-generating component (like a

button) from the event-handling code. This means the event source doesn't need to worry about what happens when the event occurs, keeping things more organized.

2. Flexibility: Multiple listeners can be attached to the same event source, allowing you to respond to the same event in different ways across different parts of your program.

3. Scalability: As your application grows, you can add more events and listeners without needing to modify your existing event sources. This makes it easy to extend functionality without breaking what's already there.

Example of the Delegation Event Model in Action

Here's a simple example showing how it works:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
public class DelegationModelExample
{
    public static void main(String[] args)
    {
        JFrame frame = new JFrame("Delegation Event Model");
        JButton button = new JButton("Click Me");
        // Register an ActionListener to the button (the event source)
        button.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent e)
            {
                System.out.println("Button clicked! Event handled by
                ActionListener.");
            }
        });
    frame.add(button);
    frame.setSize(300, 200);
```

```
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
        }
}
```

In this example:

- The event source is the JButton. When you click it, an event is generated.
- The event listener is an anonymous class implementing ActionListener. It listens for the button click and defines what should happen (in this case, printing a message to the console).
- The event object (ActionEvent) holds details about the click.

2.3.3 Event Listener Interfaces

In Java, Event Listener Interfaces play a key role in how events are managed, especially in graphical applications. These interfaces allow different objects (called listeners) to respond to various user interactions, like clicking a button, moving the mouse, or pressing a key. Essentially, event listener interfaces define the actions that should happen when specific events occur.

An event listener interface is like a contract that a class signs up for. When a class implements a listener interface, it promises to handle a specific type of event. For example, if a class implements ActionListener, it agrees to handle "action" events, such as when a button is clicked. When that event occurs, the event source (the component generating the event) will notify the listener, which will then take appropriate action.

2.3.3.1 Common Event Listener Interfaces

Java provides several listener interfaces, each designed to handle specific types of events:

1. ActionListener: This is used to handle actions like button clicks. It contains a method called actionPerformed(ActionEvent e) that runs when the action occurs.
2. MouseListener: This interface handles mouse events such as clicks and releases. It contains methods like mouseClicked(MouseEvent e) and mousePressed(MouseEvent e) to react to different types of mouse interactions.
3. KeyListener: This interface is responsible for keyboard events. It includes methods like keyPressed(KeyEvent e) and keyReleased(KeyEvent e) to handle key presses and releases.
4. WindowListener: This one deals with events related to windows, such as opening, closing, or minimizing them. Methods like windowClosing(WindowEvent e) let you define what should happen when a window is about to close.

2.3.3.2 How Event Listener Interfaces Work

1. Implementation: A class implements the listener interface to handle a specific event. This means it must define the methods that correspond to that interface.

2. Registration: The listener is then registered with an event source, such as a button or a window. This step connects the listener to the event it will handle.

3. Event Handling: When the event occurs, the listener's method (like actionPerformed() for button clicks) is called automatically, allowing the listener to respond accordingly.

2.3.4 Event Classes

In Java, event classes are like messengers that carry information about different interactions between users and a graphical user interface (GUI). These interactions could be anything from clicking a button to pressing a key or moving the mouse. Event classes capture all the details about these actions and pass them on to the appropriate listener, which then decides how to respond.

Event classes define the types of events that can happen in an application. When an event occurs, like a button click, an object of the corresponding event class is created. This object contains information such as what triggered the event (the source), the type of event, and any other relevant data (like mouse position or key pressed). This event object is then passed to the listener responsible for handling it.

2.3.4.1 Common Types of Event Classes

There are various event classes in Java, each designed to handle specific types of user actions:

1. ActionEvent: This event class captures actions like button clicks or menu selections. When a button is clicked, for instance, an ActionEvent object is created and passed to the listener to handle what should happen next.

2. MouseEvent: When a user interacts with the mouse (such as clicking or moving it), a MouseEvent is generated. This class holds details like the mouse's x and y coordinates and which button was clicked.

3. KeyEvent: If a user presses or releases a key, a KeyEvent is created. It stores details like which key was pressed and whether any modifier keys (like Shift or Ctrl) were held down.

4. WindowEvent: When a window is opened, closed, or minimized, a Window-Event is triggered, and it carries the necessary information to manage those window-related actions.

2.3.4.2 How Event Classes Work

1. Event Creation: When a user interacts with a component, an event object is created from the relevant event class. For example, clicking a button creates an ActionEvent.

2. Event Passing: This event object is passed to the listener that's been set up to handle that type of event. The listener then takes control, deciding what should happen next.

3. Processing the Event: The listener reads the event details and carries out the desired action. For example, if you click a button to submit a form, the listener might process the form data.

2.3.4.3 Importance of Event Classes

Simplify Event Handling: Event classes organize all the details about user actions in one place, making it easier to handle them.

Flexibility: These classes allow developers to respond differently to various types of events in a unified way.

Modular Design: With event classes, it's easy to break down and manage complex user interactions, keeping the application organized and scalable.

2.3.4.4 AWT Event Class and Hierarchy

When developing graphical user interfaces (GUIs) in Java, the Abstract Window Toolkit (AWT) plays a key role, especially in handling user interactions through events. At the heart of this framework lies the AWT Event Class, which serves as a foundation for all event-related activities in AWT. Let's break down what this class is and how its hierarchy is structured in an easy-to-understand way.

The AWT Event Class is part of the java.awt.event package and acts as the base class for all events in AWT. Think of it as a blueprint that captures essential information about what happens in your GUI, such as which component was interacted with and the type of interaction that occurred. By building on this class, Java can create more specific event classes for different user actions.

The AWT Event Hierarchy

The hierarchy of AWT events is organized into a clear structure that helps categorize different types of user interactions. Here's a simplified view of this hierarchy:

1. EventObject Class: This is the top-level class in the hierarchy. It serves as the parent for all events in AWT and contains basic details like the source, which indicates the component that triggered the event.

2. AWTEvent Class: This class extends the EventObject and adds more features. It includes information like the event type, which helps differentiate between various events through numeric identifiers.

3. Specific Event Classes: Under the AWTEvent class, there are several special-

ized classes, each representing a different kind of event:

- ActionEvent: This captures actions like button clicks or menu selections, allowing the application to respond when users interact with these components.
- MouseEvent: This handles all mouse-related actions, such as clicks and movements. It keeps track of where the mouse is and what buttons were pressed.
- KeyEvent: This class deals with keyboard interactions, recording which keys were pressed or released.
- WindowEvent: This class manages events related to window actions, such as opening or closing a window.

Each specific event class comes with constants that represent various actions. For example, the MouseEvent class has constants to identify different mouse activities, making it straightforward for developers to recognize what action took place.

Importance of AWT Event Class and Hierarchy

1. Organized Event Handling: The structured hierarchy makes it easy for developers to manage different types of events, streamlining the process of creating responsive applications.
2. Reusability: Developers can extend these event classes to create custom events tailored to their specific needs, which promotes modular and reusable code.
3. Improved User Interaction: By understanding this hierarchy, developers can craft applications that react seamlessly to user inputs, making for a richer user experience.

2.3.4.5 Custom Event Handling

Custom event handling in Java allows developers to create specialized responses to user interactions beyond the standard events provided by the AWT or Swing libraries. By defining custom events, programmers can tailor their applications to meet specific requirements and enhance user experience.

In Java, custom event handling involves creating new event classes and listeners to manage events that are unique to an application's context. This capability enables developers to respond to specific actions or changes within their applications, allowing for a more interactive and personalized user experience.

Steps to Implement Custom Event Handling

1. Define a Custom Event Class: To create a custom event, the first step is to define a new class that extends java.util.EventObject. This class should include any additional information relevant to the event. For example, if you're creating an event for a temperature sensor, you might include attributes

like the temperature value and a timestamp.

2. Create a Listener Interface: Next, define an interface that declares methods to handle the custom event. This interface should specify what actions should be taken when the event occurs.

3. Implement the Listener: Any class that wants to respond to the custom event must implement the listener interface. This involves providing concrete definitions for the methods declared in the interface.

4. Register the Listener: In the class that generates the events, maintain a list of registered listeners. When an event occurs, notify all registered listeners by calling their corresponding methods.

Benefits of Custom Event Handling

1. Flexibility: Custom event handling allows for tailored responses to specific user actions, enhancing the application's interactivity and user experience.

2. Modularity: By separating event generation from handling, developers can create modular code that is easier to manage and maintain.

3. Reusability: Custom events and listeners can be reused across different parts of the application or even in different projects, promoting code efficiency.

2.3.5 Advanced Event Handling Techniques

Advanced event handling techniques in Java enhance the responsiveness and functionality of applications, enabling developers to create sophisticated user interactions. These techniques extend beyond basic event handling, allowing for more complex user interfaces and better management of event-driven programming.

1. Event Filtering

Event filtering allows developers to intercept events before they reach their target components. This technique is useful for scenarios where you want to perform some checks or pre-processing on events. For instance, you can use event filters to restrict certain actions or log user interactions.

In Java, event filtering can be achieved using the EventFilter interface, which is part of the JavaFX library. By implementing this interface, you can decide whether to consume or propagate an event based on specific conditions.

2. Using Anonymous Classes for Listeners

Java allows the use of anonymous classes to implement event listeners directly where they are needed. This technique reduces the need for separate classes or lengthy implementations, making your code cleaner and more concise.

3. Lambda Expressions

With the introduction of Java 8, lambda expressions provide a more elegant way to

implement event listeners. They simplify the syntax and improve readability by allowing you to express instances of single-method interfaces more succinctly.

4. Multiple Event Sources

Handling events from multiple sources can be achieved by creating a single listener that responds to various components. This approach reduces code duplication and centralizes event management.

5. Custom Event Objects

While earlier sections focused on defining custom events, advanced techniques include creating rich custom event objects that carry additional data. This allows for more context during event handling, making it easier to implement logic based on the e vent's attributes.

6. Asynchronous Event Handling

In complex applications, handling events asynchronously can enhance performance. By using background threads or the Java Executor framework, you can offload time-consuming tasks from the event dispatch thread, keeping the UI responsive.

Recap

- Applets in Java are small programs that can be embedded in web pages to make them interactive.
- By combining applets and event handling, developers can create dynamic web components that interact with users, making websites more engaging.
- Java applets undergo a series of steps, called lifecycle methods, that manage how they run from start to finish.
- These methods - init(), start(), stop(), and destroy() - are automatically triggered by the browser or applet viewer at different points in the applet's life.
- Creating a Java applet begins by writing a Java class that extends the java. applet.Applet class.
- In Java, an event is an object that describes a change in the state of a source, like a button press or a window being closed.
- A listener is an object that "listens" for events and defines how to respond to those events when they occur.
- The Delegation Event Model is the foundation of how Java handles events in graphical user interfaces (GUIs).

- Event Listener interfaces allow different objects (called listeners) to respond to various user interactions, like clicking a button, moving the mouse, or pressing a key.
- Event classes are like messengers that carry information about different interactions between users and a graphical user interface (GUI).
- When developing graphical user interfaces (GUIs) in Java, the Abstract Window Toolkit (AWT) plays a key role, especially in handling user interactions through events.
- Custom event handling involves creating new event classes and listeners to manage events that are unique to an application's context.
- Event filtering allows developers to intercept events before they reach their target components.
- Java allows the use of anonymous classes to implement event listeners directly where they are needed.
- With the introduction of Java 8, lambda expressions provide a more elegant way to implement event listeners.
- Handling events from multiple sources can be achieved by creating a single listener that responds to various components.
- While earlier sections focused on defining custom events, advanced techniques include creating rich custom event objects that carry additional data.

Objective Type Questions

1. What are Java applets embedded in?
2. Which class do Java applets extend?
3. What manages the different stages of a Java applet's lifecycle?
4. What object describes a change in the state of a source?
5. What listens for and responds to user actions in a GUI?
6. What is the foundation of Java's event-handling model?
7. Which toolkit helps Java GUIs handle user interactions?
8. What allows Java developers to intercept events before they reach components?

9. Which Java feature introduced in Java 8 simplifies event listener implementation?

10. What type of listener can respond to multiple event sources?

Answers to Objective Type Questions

1. Webpages
2. Applet
3. Methods
4. Event
5. Listener
6. Delegation
7. AWT
8. Filtering
9. Lambda
10. Single

Assignments

1. Develop a simple Java applet that demonstrates the use of lifecycle methods and event handling. Your applet should include the following components:

    1. Applet Life Cycle
    2. Event Handling
    3. Custom Event Handling
    4. Advanced Feature (Optional)

References

1. Bates, Bert, and Kathy Sierra. *Head First Java*. 2nd ed., O'Reilly Media, 2005.

2. Rajshekhar, Sharanam Shah, and Vaishali Shah. *JDBC, Servlets, and JSP Black Book*. Dreamtech Press, 2011.

3. Evans, David R., and John C. Debs. *Database Programming with JDBC and Java*. 2nd ed., O'Reilly Media, 2000.

4. Eckel, Bruce. *Thinking in Java*. 4th ed., Prentice Hall, 2006.

5. Zakhour, Sowmya, et al. *The Java Tutorial: A Short Course on the Basics*. 6th ed., Addison-Wesley, 2015.

Suggested Reading

1. Herbert, Schildt. "Java: The complete Reference 9th edition." (2014).

2. Balagurusamy, Emir. Programming in Java: A Primer. McGraw-Hill Education, 2010.

3. Sierra, Kathy, and Bert Bates. Head First Java: A Brain-Friendly Guide. " O'Reilly Media, Inc.", 2005.

Java Database Connectivity

Learning Outcomes

After the successful completion of the course, the learner will be able to:

- **define** JDBC and its role in Java applications.
- **identify** key features of JDBC like platform independence and SQL support.
- **recall** steps for establishing a JDBC database connection.
- **list** SQL operations supported by JDBC.
- **state** the benefits of using JDBC in Java.

Prerequisites

Imagine you are trying to access a library. In this library, there are many books (representing data) stored on various shelves (representing different databases). Now, to find a book, you need a system to search, retrieve, and interact with the books in an organized way. This is where a librarian comes in. The librarian serves as a bridge between you and the shelves, helping you locate and manage the books efficiently.

In the world of Java programming, **Java Database Connectivity (JDBC)** acts like this librarian. Just like how the librarian helps you interact with the books, JDBC helps a Java application interact with different databases. It simplifies the process by providing a standard method to connect to various types of databases, allowing you to search, retrieve, update, and manage data (like how you would borrow, return, or read books in the library).

The key idea here is that JDBC is a "bridge" between your Java application and a wide range of databases, making sure you can perform all necessary data operations in a smooth and efficient manner, much like a librarian would help you navigate through a complex library system.

Discussion

2.4.1 Java Database Connectivity (JDBC)

Java Database Connectivity (JDBC) is a Java-based API (Application Programming Interface) that allows Java applications to interact with databases. JDBC is a standard Java API that provides a uniform interface for connecting to a wide range of relational databases such as MySQL, PostgreSQL, Oracle, SQL Server, and many others. It is part of the Java Standard Edition (Java SE) and has been a core feature since Java 1.1.

The primary purpose of JDBC is to enable Java applications to perform database operations, such as creating, reading, updating, and deleting data (commonly known as CRUD operations). JDBC acts as a bridge between a Java application and the database, allowing developers to execute SQL queries, retrieve results, and manage database transactions in a standardized way.

2.4.1.1 Key Features of JDBC

1. **Platform Independence :** JDBC provides a consistent interface that works seamlessly with various database systems. Developers can write a single set of code that can interact with different databases without requiring significant modifications. This feature ensures flexibility and reduces the effort needed to adapt applications for diverse environments.

2. **SQL Support :** JDBC enables Java applications to leverage the SQL language for interacting with relational databases. It allows developers to perform standard database operations like SELECT for data retrieval, INSERT for adding records, UPDATE for modifying existing records, and DELETE for removing data. By seamlessly integrating SQL with Java, JDBC provides a powerful toolset for managing and querying relational databases efficiently.

3. **Database Connectivity :** With JDBC, Java applications can establish reliable connections with databases. It facilitates the exchange of SQL queries and the receipt of responses, allowing for smooth communication between the application and the database system.

4. **Data Handling :** JDBC enables Java programs to retrieve and manipulate data from databases efficiently. It provides methods to access specific records or process large datasets, making it easier to handle complex data operations within applications.

5. **Error Management :** JDBC includes built-in mechanisms for handling errors and exceptions related to SQL operations. These tools allow developers to address unexpected issues gracefully, ensuring that the application remains stable and user-friendly in case of database-related problems.

2.4.1.2 How JDBC Works

JDBC uses a driver manager to connect a Java application and the target database. The

following steps describe how JDBC works:

1. **Loading the JDBC Driver :** A JDBC driver is a specialized software component that facilitates communication between a Java application and a specific database. The driver must be loaded into the application to enable interaction with the database system before any database operations can be performed.

2. **Establishing a Connection :** Using the JDBC API, the Java application establishes a connection with the target database. This process involves specifying key details such as the database URL, username, and password to authenticate and initiate the connection.

3. **Creating a Statement :** Once the connection is established, the application creates a statement object. This object serves as a medium to execute SQL commands, such as queries or updates, within the database.

4. **Executing SQL Queries :** The statement object is used to execute SQL queries, enabling the application to perform operations like retrieving data, inserting new records, updating existing records, or deleting records from the database.

5. **Processing the Results :** After executing a query, the database sends the results back to the application. These results can then be processed, analyzed, or displayed to the user as required by the application.

6. **Closing the Connection :** Once all necessary database operations are completed, the connection should be closed. This step is crucial to release system resources, maintain database performance, and ensure the security of the application.

2.4.1.3 Benefits of Using JDBC

- **Flexibility**: JDBC allows Java applications to connect to any relational database that supports JDBC.
- **Efficiency**: It optimises performance by using native SQL for database operations.
- **Standardisation**: Being a standard API, JDBC offers a uniform approach for database connectivity, making it easier to switch between different databases.

2.4.2 Steps to Configure a JDBC Development Environment

To develop Java applications using JDBC, you need to set up your development environment correctly. This setup involves installing the necessary software and configuring your project to use the JDBC API. Here are the steps to get started:

1. Install Java Development Kit (JDK).

2. Install a database management system (DBMS).

3. Download the appropriate JDBC driver for your DBMS.

4. Set up the database by creating required schemas and tables.

5. Configure your project to include the JDBC driver in the classpath.

6. Write and test Java code to establish a database connection using JDBC.

After installing the Java Development Kit (JDK) and a database management system (DBMS), proceed with setting up JDBC by downloading the appropriate JDBC driver for your DBMS. Configure your project to include the JDBC driver in the classpath, set up the database by creating the necessary schemas and tables, and write Java code to establish a connection to the database using JDBC. Test the connection to ensure everything is working correctly.

2.4.2.1 Add JDBC Driver to the Project

- Select the JDBC Driver: Determine the JDBC driver for your specific database (e.g., MySQL, PostgreSQL, Oracle, etc.). JDBC drivers are typically available for download from the database vendor's website.
- Download the Driver: Download the appropriate JDBC driver JAR file.
- Add the Driver to Your Project: In your IDE, add the JDBC driver JAR file to your project's build path:
  - For Eclipse: Right-click on the project > Build Path > Configure Build Path > Add External JARs.
  - For IntelliJ IDEA: Right-click on the project > Open Module Settings > Libraries > + > Add JARs.
  - For NetBeans: Right-click on the project > Properties > Libraries > Add JAR/Folder.

2.4.2.2 Establish Database Connectivity

- Install the Database: Make sure the database you want to connect to is installed and running. For example, install MySQL, PostgreSQL, Oracle, or any other relational database.
- Create a Database: Create a new database or use an existing one. Take note of the database name, username, and password.
- Configure Database Access: Ensure the database is configured to accept connections from your development environment. For local development, this often involves setting the database to accept connections from localhost.

2.4.2.3 Write a JDBC Test Program

- **Create a New Java Project**: Open your IDE and create a new Java project.

- **Write the JDBC Code**: Write a simple Java program to test the JDBC connection.

Here is a basic example for connecting to a MySQL database. See Program : Example JDBC Connection establishment

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class JDBCExample {
   public static void main(String[] args) {
String jdbcURL = "jdbc:mysql://localhost:3306/your_database";
      String username = "your_username";
      String password = "your_password";
      try {
         Connection connection = DriverManager.getConnection(jdbcURL, username, password);
            System.out.println("Connected to the database successfully!");
       connection.close();
      } catch (SQLException e) {
System.out.println("Error connecting to the database: " + e.getMessage());
                  }
         }
}
```

Program 2.4.1: Example of JDBC Connection

2.4.2.6 Compile and Run the Java Program

Compile the Program: Use the IDE to compile the Java program or use the command line:

```
javac JDBCExample.java
```

Run the Program: Execute the compiled program using the IDE or command line:

```
java JDBCExample
```

2.4.2.7 Verify the Connection

Check the console output to confirm that the connection to the database was successful. If there are any errors, they will be displayed, and you should resolve them accordingly (e.g., incorrect URL, username, or password)

2.4.3 Processing SQL Statements with JDBC

Java Database Connectivity (JDBC) allows developers to interact with databases using SQL statements. Here is an outline of the steps required to process SQL statements with JDBC, along with detailed explanations of each step:

2.4.3.1 Establishing a Connection

To interact with a database, the first step is to establish a connection. This involves using the DriverManager class to connect to the database by providing the database URL, username, and password.

2.4.3.2 Connecting with DataSource Objects

Using DataSource objects is the preferred way to connect to a database in JDBC. DataSource objects provide a more flexible and portable way to create database connections. They can be configured to use connection pooling, which improves performance by reusing existing database connections.

2.4.3.3 Handling SQLExceptions

When working with databases, errors may occur, such as connection issues or incorrect SQL syntax. JDBC provides the SQLException class to handle these errors gracefully, ensuring the application continues to run or provides meaningful error messages.

2.4.3.4 Setting Up Tables

To perform database operations, you need tables. This step involves using SQL scripts or JDBC API methods to create and populate the necessary database tables. This setup is crucial for running queries and storing data.

2.4.3.5 Retrieving and Modifying Values from ResultSets

After setting up the tables, you can retrieve and modify data using SQL queries. A result set, which contains the data requested, is returned when a query is executed. We can then iterate through this result set to retrieve and manipulate the values.

2.4.3.6 Using Prepared Statements

Prepared statements provide a more efficient and secure way to execute SQL queries. They are precompiled SQL statements that can be reused multiple times with different parameters, reducing the risk of SQL injection attacks and improving performance.

2.4.3.7 Using Transactions

Transactions allow you to control when a series of SQL statements are executed. You

can group multiple operations into a single transaction and commit them only when all operations are successful, ensuring data integrity.

2.4.3.8 Using RowSet Objects

RowSet objects in JDBC are specialised versions of ResultSet objects, providing greater flexibility for managing and manipulating tabular data. Unlike standard ResultSet objects, RowSet objects can operate in both connected and disconnected modes, which makes them particularly useful for applications that need to work with data offline or need to frequently exchange data between different components. Here's an overview of the different types of RowSet objects and their use cases:

2.4.4 Types of RowSet Objects

1. JdbcRowSet

- Description: A JdbcRowSet is a connected RowSet that maintains an active connection to the database. It combines the capabilities of a ResultSet with additional features such as scrolling through data in both directions (forward and backwards) and the ability to update rows directly.
- Use Case: Ideal for applications that require a constant connection to the database for real-time data operations, such as monitoring systems or applications that frequently update database records.

Example:

```
JdbcRowSet jdbcRowSet = RowSetProvider.newFactory().createJdbcRowSet();
jdbcRowSet.setUrl("jdbc:mysql://localhost:3306/your_database");
jdbcRowSet.setUsername("your_username");
jdbcRowSet.setPassword("your_password");
jdbcRowSet.setCommand("SELECT * FROM your_table");
jdbcRowSet.execute();
while (jdbcRowSet.next()) {
System.out.println("Column Data: " + jdbcRowSet.getString("column_name"));
}
```

Program 2.4.2 Example of JdbcRowSet

2. CachedRowSet

- A CachedRowSet is a disconnected RowSet that caches data in memory. Once data is retrieved from the database, it can be manipulated without maintaining an active database connection. Changes can be synchronised

back to the database later.

- Useful for applications that need to work with data offline, such as mobile applications, or for minimizing database connections to reduce overhead.

Example:

```
CachedRowSet cachedRowSet = RowSetProvider.newFactory().
createCachedRowSet();
cachedRowSet.setUrl("jdbc:mysql://localhost:3306/your_database");
cachedRowSet.setUsername("your_username");
cachedRowSet.setPassword("your_password");
cachedRowSet.setCommand("SELECT * FROM your_table");
cachedRowSet.execute();
cachedRowSet.absolute(2); // Move to the second row
cachedRowSet.updateString("column_name", "new_value");
cachedRowSet.updateRow(); // Apply the changes
```

Program 2.4.3: Example of CachedRowSet

3. JoinRowSet

A JoinRowSet provides SQL join capabilities. It enables data to be joined from multiple RowSet objects, effectively mimicking an SQL JOIN operation without requiring a direct SQL query on the database.

Suitable for merging data from multiple sources or tables in memory without requiring additional SQL joins in the database.

Example:

```
JoinRowSet joinRowSet = RowSetProvider.newFactory().createJoinRowSet();
CachedRowSet rowSet1 = RowSetProvider.newFactory().createCachedRowSet();
CachedRowSet rowSet2 = RowSetProvider.newFactory().createCachedRowSet();
// Configure rowSet1 and rowSet2 with appropriate data...
joinRowSet.addRowSet(rowSet1, "common_column");
joinRowSet.addRowSet(rowSet2, "common_column");
```

```
while (joinRowSet.next()) {
  System.out.println("Joined Data: " + joinRowSet.getString("common_column"));
}
```

Program 2.4.4: Example of JoinRowSet

FilteredRowSet

A FilteredRowSet allows for data filtering using custom filter criteria. It implements the Predicate interface, enabling developers to specify conditions for filtering rows in the RowSet.

Ideal for applications that need to display or manipulate data based on dynamic filtering, such as search results or data views that change based on user input.

Example:

```
FilteredRowSet filteredRowSet = RowSetProvider.newFactory().createFilteredRowSet();
filteredRowSet.setUrl("jdbc:mysql://localhost:3306/your_database");
filteredRowSet.setUsername("your_username");
filteredRowSet.setPassword("your_password");
filteredRowSet.setCommand("SELECT * FROM your_table");
filteredRowSet.execute();
Predicate filter = new Predicate() {
        @Override
        public boolean evaluate(RowSet rs) {
                // Implement custom filter logic here
                 return true;
        }
                // Implement other required methods...
  };
filteredRowSet.setFilter(filter);
```

```
while (filteredRowSet.next()) {
        System.out.println("Filtered Data: " + filteredRowSet.getString("column_name"));
  }
```

Program 2.4.5: Example of FilteredRowSet

4. **WebRowSet**

A WebRowSet is a RowSet that can read and write data in XML format. It provides methods for reading data from an XML document and writing data to XML, making it useful for web applications or services that need to exchange data in a standardized format.

Suitable for web services or applications that require data exchange or persistence in XML format, such as RESTful web services.

Example: WebRowSet

```
WebRowSet webRowSet = RowSetProvider.newFactory().createWebRowSet();
webRowSet.setUrl("jdbc:mysql://localhost:3306/your_database");
webRowSet.setUsername("your_username");
webRowSet.setPassword("your_password");
webRowSet.setCommand("SELECT * FROM your_table");
webRowSet.execute();
webRowSet.writeXml(System.out); // Write the data as XML to the console
```

Program 2.4.6: Example of WebRowSet

2.4.5 Using Stored Procedures in JDBC

Stored procedures are precompiled collections of one or more SQL statements stored in the database. They act like functions, allowing you to encapsulate complex database operations into reusable blocks of code. By using stored procedures, you can simplify application logic, improve performance by reducing network traffic (as multiple SQL operations are executed on the database server), and ensure consistent execution of repetitive tasks.

JDBC provides support for stored procedures, enabling developers to create, execute, and manage them from Java applications. Here's how to use stored procedures with JDBC:

2.4.5.1 Why should we Use Stored Procedures ?

1. **Simplifies Complex Operations :** Stored procedures encapsulate complex business logic and database operations directly within the database. This reduces the need for intricate coding in the Java application, simplifying development and maintenance.

2. **Improves Performance :** Stored procedures minimize network traffic by allowing multiple SQL statements to be executed in a single call. Additionally, they benefit from precompilation by the database, leading to faster execution times.

3. **Enhances Security :** By restricting direct access to the database and its structures, stored procedures enhance security. They allow users to perform only specific, predefined operations, safeguarding the underlying data.

2.4.6 Steps to Use Stored Procedures in JDBC

2.4.6.1 Creating a Sample Table and Stored Procedure in MySQL

Before creating a stored procedure, you need a sample table in your database. Here's an example that demonstrates how to set up a table and create a stored procedure:

1. Create a Sample Table

```
CREATE TABLE employees (
    emp_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    department VARCHAR(50),
    salary DECIMAL(10, 2)
);
```

2. Insert Sample Data into the Table

```
INSERT INTO employees (name, department, salary)
VALUES
('Alice', 'HR', 50000.00),
('Bob', 'Engineering', 75000.00),
('Charlie', 'Marketing', 60000.00);
```

3. Create a Stored Procedure

The following stored procedure retrieves employees from a specific department:

```
CREATE PROCEDURE GetEmployeesByDepartment(IN dept_name VARCHAR(50))
```

```
BEGIN
    SELECT emp_id, name, salary
    FROM employees
    WHERE department = dept_name;
END
```

4. Call the Stored Procedure

Use the following SQL command to call the procedure and retrieve employees from the "Engineering" department:

```
CALL GetEmployeesByDepartment('Engineering');
```

5. Output Example

The result will display the emp_id, name, and salary of employees in the specified department.

Establish a Database Connection in Java Use JDBC to establish a connection to the database where the stored procedure is defined:

```
Connection conn = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/your_database", "your_username", "your_password");
```

Prepare the CallableStatement: To call the stored procedure, you use the CallableStatement interface. It allows you to call stored procedures in a standard way across different databases.

In JDBC, a CallableStatement is used to execute stored procedures in a database. Stored procedures are predefined SQL code blocks stored in the database that can be executed multiple times, often to encapsulate complex database operations. The CallableStatement interface extends PreparedStatement and allows Java applications to call these stored procedures. You create a CallableStatement by using the prepareCall() method on a Connection object and providing the appropriate SQL syntax for calling the procedure. This enables applications to execute the stored procedure and handle input and output parameters.

The executeUpdate() method is commonly used with CallableStatement or PreparedStatement to execute SQL statements that modify the database, such as INSERT, UPDATE, or DELETE operations. It returns an integer value representing the number of rows affected by the operation. This method is crucial for determining whether a modification query was successful and how many rows were impacted, making it particularly useful for transactional operations or batch processing where result tracking is important.

```
CallableStatement stmt = conn.prepareCall("{CALL GetEmployeeName(?, ?)}");
```

The {CALL GetEmployeeName(?, ?)} syntax is the JDBC escape syntax for calling

stored procedures. The question marks (?) represent placeholders for input or output parameters.

Set Input and Register Output Parameters Use the setXXX methods to set input parameters and the registerOutParameter method to register output parameters:

```
stmt.setInt(1, 101); // Set the input parameter (employee ID)

stmt.registerOutParameter(2, java.sql.Types.VARCHAR); // Register the output parameter (employee name)
```

1. Execute the CallableStatement Execute the stored procedure using the execute method:

```
stmt.execute();
```

2. Retrieve Output Parameters After execution, retrieve the output parameter values using the appropriate getXXX method:

```
String employeeName = stmt.getString(2); // Get the output parameter (employee name)

System.out.println("Employee Name: " + employeeName);
```

3. Close Resources Always close the CallableStatement and Connection objects to release database resources:

```
stmt.close();

conn.close();
```

4. Using a Stored Procedure in JDBC : Here's a complete example that demonstrates how to call a stored procedure from a Java application using JDBC:

Example Program: Fetching Data from a Table

Sample Table: employees

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    department VARCHAR(50),
    salary DECIMAL(10, 2)
);
INSERT INTO employees (id, name, department, salary) VALUES
(1, 'John Doe', 'IT', 70000.00),
(2, 'Jane Smith', 'HR', 65000.00),
(3, 'Mike Brown', 'Finance', 72000.00);
```

Java Program: Fetch and Display Data

```
import java.sql.*;
public class FetchDataExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/your_database";
        String username = "your_username";
        String password = "your_password";
        try {
            // Establish database connection
            Connection conn = DriverManager.getConnection(url, username, password);
// Query to fetch employees with salary greater than a specific amount
        String query = "SELECT id, name, department, salary FROM employees WHERE
salary > ?";
            // Prepare the statement
            PreparedStatement stmt = conn.prepareStatement(query);
            stmt.setDouble(1, 68000.00); // Set the salary threshold
            // Execute the query
            ResultSet rs = stmt.executeQuery();
            // Display the results
            System.out.println("Employees with salary greater than 68000:");
            System.out.println("ID\tName\t\tDepartment\tSalary");
            System.out.println("-------------------------------------------------");
            while (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
                String department = rs.getString("department");
                double salary = rs.getDouble("salary");
             System.out.printf("%d\t%-15s%-15s%.2f\n", id, name, department, salary);
            }
```

```
            // Close the resources
            rs.close();
            stmt.close();
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Steps to Run the Program

1. Set Up the Database:

- Create the employees table and insert sample data as shown above.
- Update the url, username, and password variables with your database credentials.

2. Compile and Run:

- Save the Java code in a file named FetchDataExample.java.
- Compile it using javac FetchDataExample.java.
- Run the program using java FetchDataExample.

Expected Output

```
Employees with salary greater than 68000:
ID    Name          Department    Salary
-------------------------------------------------
1     John Doe      IT            70000.00
3     Mike Brown    Finance       72000.00
```

Recap

- **JDBC Overview**: JDBC (Java Database Connectivity) is an API that allows Java applications to connect and interact with relational databases.
- **Platform Independence**: JDBC provides a uniform interface to connect Java applications to different databases like MySQL, Oracle, SQL Server, etc.
- **SQL Support**: JDBC allows executing SQL commands (such as SELECT, INSERT, UPDATE, DELETE) from Java applications.
- **Driver Management**: JDBC uses a driver manager to load the appropriate database driver and manage database connections.
- **Steps in JDBC Connection**:
  - Load the appropriate JDBC driver.
  - Establish a connection to the database using a URL, username, and password.
  - Create a statement object for executing SQL queries.
  - Execute SQL queries (e.g., retrieving or updating data).
  - Process the results from the queries.
  - Close the connection after operations are completed.
- **Error Handling**: JDBC includes mechanisms for managing SQL exceptions and errors during database operations.
- **Efficiency**: JDBC supports database connection pooling and transaction management for better performance.

Objective Type Questions

1. What is the primary purpose of JDBC in Java applications?
2. Which Java package contains the classes and interfaces for JDBC?
3. Name the interface used to establish a connection to a database in JDBC.
4. How can you load a specific database driver in a JDBC program?
5. What method is used to establish a connection to a database in JDBC?

6. What is the SQL command for retrieving data from a database using JDBC?
7. Which interface is used to execute SQL queries in JDBC?
8. How do you close a database connection in JDBC to free resources?
9. What is the difference between Statement and PreparedStatement in JDBC?
10. What is the purpose of ResultSet in JDBC?
11. Which exception is commonly thrown when a database operation fails in JDBC?
12. What is a DataSource in JDBC?
13. Describe the role of the DriverManager class in JDBC.
14. How can you retrieve the number of columns in a ResultSet in JDBC?
15. What are the main benefits of using PreparedStatement over Statement?
16. How do you handle transactions in JDBC?
17. Explain how to use a stored procedure in JDBC.
18. What is a CallableStatement used for in JDBC?
19. What does the method executeUpdate() return in JDBC?
20. How can you fetch metadata about a database in JDBC?

Answers to Objective Type Questions

1. The primary purpose of JDBC is to enable Java applications to interact with databases.
2. The java.sql package contains the classes and interfaces for JDBC.
3. The Connection interface is used to establish a connection to a database in JDBC.
4. You can load a specific database driver using Class.forName("driver_class_name").
5. The DriverManager.getConnection() method is used to establish a connection to a database.

6. The SQL command for retrieving data is SELECT.

7. The Statement interface is used to execute SQL queries in JDBC.

8. You close a database connection using the close() method on the Connection object.

9. Statement is used for simple queries without parameters, while PreparedStatement is used for precompiled queries with parameters.

10. The purpose of ResultSet is to hold the data retrieved from a database after executing a query.

11. The SQLException is commonly thrown when a database operation fails in JDBC.

12. A DataSource is an interface that provides a more flexible way to obtain database connections compared to DriverManager.

13. The DriverManager class manages a list of database drivers and establishes a connection to a database.

14. You can retrieve the number of columns in a ResultSet using the getMetaData().getColumnCount() method.

15. Benefits of using PreparedStatement include improved performance and protection against SQL injection.

16. You handle transactions in JDBC using Connection.setAutoCommit(false), followed by commit() or rollback().

17. To use a stored procedure in JDBC, you create a CallableStatement and call it using the appropriate syntax.

18. A CallableStatement is used for executing stored procedures in JDBC.

19. The executeUpdate() method returns the number of rows affected by the SQL statement.

20. You can fetch metadata about a database using the DatabaseMetaData interface.

Assignments

1. **Write a Java program** to establish a connection with a MySQL database using JDBC. Use the appropriate driver, and print a success message once the connection is established.

2. **Explain the role of the DriverManager** class in JDBC. How does it manage the database drivers, and what steps are involved in using it to connect to a database?

3. **Create a JDBC program** to perform a simple CRUD operation (Create, Read, Update, Delete) on a database table. Implement at least one SQL query for each operation.

References

1. Bates, Bert, and Kathy Sierra. *Head First Java*. 2nd ed., O'Reilly Media, 2005.

2. Rajshekhar, Sharanam Shah, and Vaishali Shah. *JDBC, Servlets, and JSP Black Book*. Dreamtech Press, 2011.

3. Evans, David R., and John C. Debs. *Database Programming with JDBC and Java*. 2nd ed., O'Reilly Media, 2000.

4. Eckel, Bruce. *Thinking in Java*. 4th ed., Prentice Hall, 2006.

5. Zakhour, Sowmya, et al. *The Java Tutorial: A Short Course on the Basics*. 6th ed., Addison-Wesley, 2015.

Suggested Reading

1. Fisher, Maydene, Jon Ellis, and Jonathan Bruce. *JDBC™ API Tutorial and Reference*. Addison-Wesley Professional, 2001.

2. Oracle. *JDBC API Documentation*. Oracle, https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/. Accessed 30 Sept. 2024.

3. Schildt, Herbert. *Java: The Complete Reference*. 10th ed., McGraw-Hill Education, 2017.

4. Horstmann, Cay S. *Core Java Volume II: Advanced Features*. 11th ed., Pearson, 2019.

5. McLaughlin, Brett. *Building Java Enterprise Applications*. O'Reilly Media, 2002.

സർവ്വകലാശാലാഗീതം

വിദ്യയാൽ സ്വതന്ത്രരാകണം
വിശ്വപൗരരായി മാറണം
ഗ്രഹപ്രസാദമായ് വിളങ്ങണം
ഗുരുപ്രകാശമേ നയിക്കണേ

കൂരിരുട്ടിൽ നിന്നു ഞങ്ങളെ
സൂര്യവീഥിയിൽ തെളിക്കണം
സ്നേഹദീപ്തിയായ് വിളങ്ങണം
നീതിവൈജയന്തി പാറണം

ശാസ്ത്രവ്യാപ്തിയെന്നുമേകണം
ജാതിഭേദമാകെ മാറണം
ബോധരശ്മിയിൽ തിളങ്ങുവാൻ
ജ്ഞാനകേന്ദ്രമേ ജ്വലിക്കണേ

കുരീപ്പുഴ ശ്രീകുമാർ

SREENARAYANAGURU OPEN UNIVERSITY

Regional Centres

Kozhikode
Govt. Arts and Science College
Meenchantha, Kozhikode,
Kerala, Pin: 673002
Ph: 04952920228
email: rckdirector@sgou.ac.in

Thalassery
Govt. Brennen College
Dharmadam, Thalassery,
Kannur, Pin: 670106
Ph: 04902990494
email: rctdirector@sgou.ac.in

Tripunithura
Govt. College
Tripunithura, Ernakulam,
Kerala, Pin: 682301
Ph: 04842927436
email: rcedirector@sgou.ac.in

Pattambi
Sree Neelakanta Govt. Sanskrit College
Pattambi, Palakkad,
Kerala, Pin: 679303
Ph: 04662912009
email: rcpdirector@sgou.ac.in

PROGRAMMING IN JAVA

COURSE CODE: B21CA01SE

Sreenarayanaguru Open University
Kollam, Kerala Pin- 691601, email: info@sgou.ac.in, www.sgou.ac.in Ph: +91 474 2966841

ISBN 978-81-984969-2-8