Fundamentals of Java Programming

ECAP392

Edited by:
Dr. Sartaj Singh

LOVELY PROFESSIONAL UNIVERSITY

Fundamentals of Java Programming

Edited By
Dr. Sartaj Singh

Title: Fundamentals of Java Programming

Author's Name: Harjinder Kaur, Dr. Neeraj Mathur

Published By : Lovely Professional University

Publisher Address: Lovely Professional University, Jalandhar Delhi GT road, Phagwara - 144411

Printer Detail: Lovely Professional University

Edition Detail: (I)

ISBN:

`

Content

Unit 01: Introduction

CONTENTS

Objectives

After this unit you will be able to:

- Understand the importance of Java programming.
- Identify the various features of Java.
- Know the importance of different parts of the main method.
- Learn the different types of access modifiers.

Introduction

Java is an important object-oriented programming language used in the software industry today. Object-oriented programming is also known as OOP. Objects are the basic elements of object-oriented programming. OOPS (Object Oriented Programming System) is used to describe a computer application that comprises multiple objects connected. It is a type of programming language in which the programmers not only define the data type of a data structure (files, arrays, and so on) but also define the behavior of the data structure. Therefore, the data structure becomes an object, which includes both data and functions.

Did you Know? Sun Microsystems originally wanted to name Java "OAK". But it could not do so as Oak Technologies already took that name. Other names that were suggested were "Silk and "DNA". Ultimately, the name "Java" was selected because it gave the Web a "jolt", and Sun intended to abstain from names that sounded very technical.

1.1 Introduction To Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm that is based on the concept of "objects." It is widely used in Java and many other programming languages. In OOP, the key idea is

to model real-world entities as objects, which have attributes (data) and behaviors (methods). Java is a versatile and object-oriented programming language, and here's an introduction to key OOP concepts in Java:

Class and Object:

Class: A class is a blueprint or a template for creating objects. It defines the properties (attributes) and behaviors (methods) that an object of that type will have.

Object: An object is an instance of a class. It is a tangible entity that represents a real-world concept and can be manipulated using the methods defined in its class.

Encapsulation:

Encapsulation is the bundling of data (attributes) and methods that operate on that data into a single unit, i.e., a class. It restricts access to some of the object's components, providing data protection and a clear interface for interacting with the object.

Inheritance:

Inheritance is a mechanism that allows a class to inherit properties and behaviors from another class. The class that is inherited from is called the superclass (or base class), and the class that inherits is called the subclass (or derived class).

Polymorphism:

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent different types, and it comes in two forms: compile-time polymorphism (method overloading) and runtime polymorphism (method overriding).

1.2 Features of Java

Let us understand Java better by understanding its important features. The following features of Java make it an important programming language:

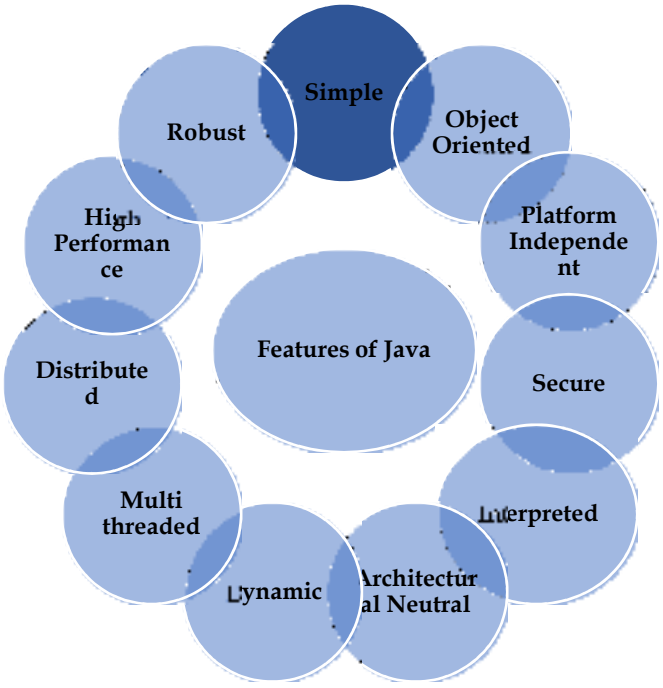


Figure 1: Features of Java Programming

Platform Independent: The write-once-run-anywhere approach towards programming is one of the key features of Java that makes it a powerful programming language. The programs written on one platform can run on any platform, irrespective of the hardware. But the hardware platform used to execute Java programs must have the Java Virtual Machine (JVM).

Simple: Various features make Java a simple language, which can be easily learned and effectively used. Java does not use pointers explicitly, thereby making it easy to write and debug programs. Java is capable of delivering a bug-free system due to its strong memory management. It also has an automatic memory allocation and de-allocation system in place.

Object-Oriented: To qualify as an object-oriented language, a language must exhibit four characteristics:

(a) *Inheritance*: It is the technique of creating new classes by making use of the behavior of the existing classes. This is done by extending the behavior of the existing classes just by adding additional features as required, thus bringing in the reusability of existing code.

(b) *Encapsulation:* It refers to the bundling of data along with the methods that act on that data.

(c) *Polymorphism:* Polymorphism, which means one name multiple forms, is the ability of a reference variable to change behavior according to the instance of the object that it holds.

(d) *Dynamic binding:* It is the method of providing maximum functionality to a program by resolving the type of object at runtime.

Robust: Java supports some features such as automatic garbage collection, strong memory allocation, powerful exception handling, and a type checking mechanism. The compiler checks the program for errors and the interpreter checks for any run time errors, thus preventing the system crash. These features make Java robust.

Distributed: The protocols like HTTP and FTP, which are extensively used over the Internet are developed using Java. Programmers who work on the Internet can call functions with the help of these protocols and can secure access to the files that are present on any remote machine on the Internet. This is made possible by writing codes on their local system itself.

Portable: The feature 'write-once run anywhere, anytime' makes Java portable, provided that the system has JVM. Java standardizes the data size, irrespective of the operating system or the processor. These features make Java a portable language.

Dynamic: A Java program also includes a significant amount of runtime information that is used to verify and resolve access to objects at runtime. This allows the code to link dynamically securely and appropriately.

Secure: Memory pointers are not explicitly used in Java. All programs in Java are run under a Java execution environment. Therefore, while downloading an applet program using the Internet, Java does not allow any virus or other harmful code to access the system as it confines it to the Java execution environment.

Performance: In Java, a program is compiled into an intermediate representation, which is called Java bytecode. This code can be executed on any system that has a JVM running on it. Earlier attempts to achieve cross-platform operability accomplished it at the cost of performance. Java bytecode is designed in such a manner that it is easy to directly translate the bytecode into the native machine code by using a just-in-time compiler. This helps in achieving high performance.

Multithreaded: The primary objective that led to the development of Java was to meet the real-world requirement of creating interactive and networked programs. To accomplish this, Java provides multithreaded programming, which permits a programmer to write programs that can do many things simultaneously.

Interpreted: Java programs can be directly run from the source code. The source code is read by the interpreter program and translated into computations. The source code generated is platform-independent. As an interpreted language, Java has an error debugging facility that can trace any error occurring in the program.

Architecture Neutral: These features of Java have made it a popular programming language. Java is also known as an architectural neutral language. In this era of networks, easy migration of applications to different computer systems having different hardware architectures and/or operating systems is necessary. The Java compiler generates an object file format that is architecture-neutral. This permits a Java application to execute anywhere on the network and many different processors, given the presence of the Java runtime system.

1.3 Java Classes

A class in Java represents a set of objects that share common characteristics and behaviors. It serves as a blueprint or prototype from which objects are created.

 **Example,** "Student" can be a class, and an individual student like "Ravi" would be an object of that class.

Properties of Java Classes:

Not a Real-world Entity: A class is an abstraction and not a tangible, real-world entity. It provides a conceptual framework for creating objects.

No Memory Occupation: A class itself does not occupy memory. Memory is allocated when objects (instances) of the class are created.

Group of Variables and Methods: A class consists of variables (data members) and methods (functions) that define the behavior of objects belonging to that class.

Contains:

Data Members: Variables that represent the attributes or properties of objects.

Methods: Functions that define the behavior of objects.

Constructor: Special methods used for initializing objects when they are created.

Nested Class: A class defined within another class.

Interface: A collection of abstract methods that can be implemented by classes.

Syntax of declaring a class:

```
access_modifier class <class_name>
{
   data member;
   method;
   constructor;
   nested class;
   interface;
}
```

 **Example**

```
public class Student {
   // Data members
   String name;
   int age;
   double grade;
   // Constructor
   public Student(String name, int age, double grade) {
      this.name = name;
      this.age = age;
      this.grade = grade;
   }
```

```
    // Method
    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Grade: " + grade);
    }
}
```

Creating an Object from the Class:

```
public class Main {
    public static void main(String[] args) {
        // Creating an object of the Student class
        Student ravi = new Student("Ravi", 20, 85.5);

        // Calling a method of the Student class
        ravi.displayInfo();
    }
}
```

This example illustrates the basic structure of a class, including data members, a constructor, and a method. Objects are then created from the class, allowing you to work with and manipulate instances of the defined blueprint.

Cautions: In Java, the class name and the file name must be the same. Also, since Java is case-sensitive, one should always provide the same case letters for both file name and class name.

After compiling and interpreting, a program is run. To run a program, follow the instructions given below:

1. Go to the command prompt.
2. Select the drive (and the folder) where the program file is saved.
3. Compile, Interpret and Run the program.

Example: Type the following program and save it in a file with the name "FirstProgram.java"

```
class FirstProgram
{
public static void main(String args[ ])
{
System.out.println("My first program in Java");
}
}
```

The process to compile and run the above-given example is described in the following steps

In the command prompt, type the directory name, and the file name.

Example: c:\jdk\programs>javac FirstProgram.java

C:\jdk\programs > java FirstProgram

In Java, a class declaration typically includes several components, listed in order:

Modifiers: A class can have modifiers like public, which determines the visibility of the class (whether it can be accessed from other classes or packages).

Class Keyword: The class keyword is used to declare a class.

Class Name: The name of the class, following Java naming conventions. It should begin with an initial uppercase letter.

Superclass (if any): If the class extends another class (inherits from a superclass), the name of the parent class follows the extends keyword. A class can extend only one parent.

Interfaces (if any): If the class implements interfaces, the names of those interfaces are listed after the implements keyword. A class can implement multiple interfaces.

Body: The class body, enclosed within curly braces {}, contains the fields, methods, and other components of the class.

In addition to these components, classes in Java commonly include:

Constructors: Special methods used for initializing new objects. Constructors have the same name as the class and are invoked when an object is created.

Fields: Variables that represent the state of the class and its objects. These can include instance variables and class variables.

Methods: Functions that define the behavior of the class and its objects. Methods can perform various actions and may return values.

In real-time applications, various types of classes are used, including:

Nested Classes: Classes defined within another class. They can be static or non-static.

Anonymous Classes: Classes without a name, often used for one-time use, especially in event handling.

Lambda Expressions: Introduced in Java 8, lambda expressions provide a concise way to express instances of single-method interfaces (functional interfaces).

Structure of java program

```
// Modifiers ClassKeyword ClassName Superclass(if any) Interfaces(if any) Body
public class MyClass extends ParentClass implements Interface1, Interface2 {
  // Fields

  // Constructors

  // Methods

  // Nested Classes

  // Anonymous Classes

  // Lambda Expressions
}
```

This structure provides a comprehensive overview of the components that can be part of a Java class declaration.

Task: Write a simple Java program to print "Java".

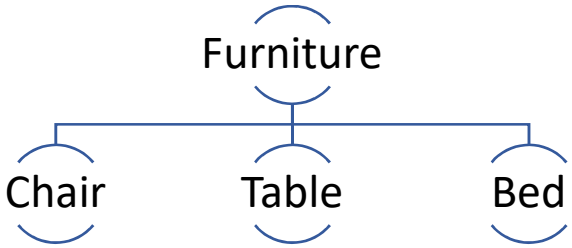
1.4 Objects

In Java, an object serves as the fundamental unit of Object-Oriented Programming, embodying real-life entities. Objects are instances of classes, created to leverage the attributes and methods defined by a class. In a Java program, numerous objects are generated, interacting by invoking methods. The composition of an object includes:

State: This is encapsulated by the attributes of an object, portraying its properties. The state represents the current values of the object's variables.

Behavior: The behavior of an object is manifested through its methods. These methods define the actions or responses that an object can perform or exhibit when interacting with other objects.

Identity: Each object possesses a distinct identity, offering a unique name or reference. This identity enables one object to engage with other objects, facilitating communication and collaboration within the program.



Objects of furniture class

In summary, Java objects encapsulate state through attributes, exhibit behavior through methods, and possess a unique identity, promoting effective interaction and modeling of real-world entities in the programming paradigm.

1.5 Main Method

The main method is a pivotal component in Java programming, often the first point of entry for executing a Java program. Its significance lies in its ability to contain code for execution or to invoke other methods. This key method can be situated within any class belonging to a program.

In more intricate programs, it is common to designate a class exclusively for the main method. The class holding the main method can bear any name, with "Main" being a customary choice. It is worth noting that the main method is recognized as the starting point for program execution.

An interesting feature is the flexibility in naming the String array argument. For instance, the conventional "args" can be altered to "myStringArgs" without affecting functionality. Alternatively, the String array argument can be expressed as "String... args" or "String args[]". This adaptability allows developers to choose a naming convention that aligns with their preferences or project requirements.

Java's main method comprises six elements: three reserved words, the main method name, a reference type, and a variable name:

public: The main method requires a public access modifier to allow the Java Runtime Environment (JRE) to access and execute it.

static: The main method is static, indicating that it can be invoked without creating an instance of the class beforehand. This is essential because when a Java program starts, there is no object of the class present.

void: The main method's return type is void, signifying that it does not return any value upon completion. Unlike some programming languages that return a status code, Java's main function doesn't provide a return value.

main: The main method is the entry point for a standalone Java application. When the JVM starts an application, the main method is the function that gets invoked.

String[] args: An array of configuration parameters can be passed into the main method as arguments. These parameters are typically named args.

args: The configuration parameters passed into the main method are conventionally named args.

In the following example, the main method is declared without the public modifier:

 Example

```
class abc {
    static void main(String[] args) {
      System.out.println("Great Day Everyone ");
   }
}
```

Output

```
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> javac abc.java
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> java abc
```

Error: Main method not found in class abc, please define the main method as: public static void main(String[] args)

or a JavaFX application class must extend javafx.application.Application

In the following example, the main method is declared without the static modifier:

 **Example**

```
public class abc {
   public void main(String[] args) {
      System.out.println("Great Day Everyone");
   }
}
```

Output

```
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> javac abc.java
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> java abc
```

Error: Main method is not static in class abc, please define the main method as: public static void main(String[] args)

Compiling and running this program would result in an error because the main method isn't static. The correct declaration should be:

 **Example**

```
public class abc {
public static void main(String[] args) {
System.out.println("Great Day Everyone ");
   }
}
```

Output

```
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> javac abc.java
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> java abc
```

Great Day Everyone

The main method is static to allow it to be used as the entry point for an application before any other Java code has run or instances have been created. If the main method were not static, it would require code to have already run for it to be invoked.

It's also crucial to note that the main method must have the exact signature: public static void main(String[] args).

Additionally, the main method accepts a single argument of type String array (String[] args). These command line arguments can be utilized to affect the program's operation or pass information at runtime.

Here's an example illustrating how to print command line arguments:

 Example

```
public class abc {
   public static void main(String[] args) {
      for (String s : args) {
         System.out.println(s);
      }
   }
}
```

Output

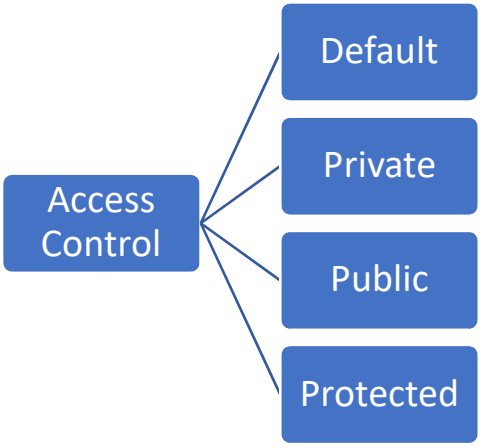
```
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> javac abc.java
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> java abc welcome all
welcome
all
```

This code iterates through the command line arguments and prints each one.

 **Cautions:** The main method conventionally serves as a singular entry point for a program, potentially constraining its flexibility by centralizing the initiation logic in one location.

1.6 Access Control

Access control in Java refers to the mechanisms and rules that govern the visibility and availability of classes, methods, variables, and other program elements. Java provides several access control modifiers to specify the level of access that classes and their members have. Access modifiers are crucial elements in Java programming that enable developers to control the scope and visibility of various program elements such as classes, constructors, variables, methods, and data members. By using access modifiers, developers can enhance security, manage accessibility, and achieve encapsulation in their code.



Let's explore the types of access modifiers available in Java:

Default (Package-Private):

No specific keyword is required for the default access modifier.

Members with the default modifier are accessible within the same package but not outside it.

Syntax

```
class MyClass {
   // Default access modifier
   int myDefaultVariable;
   void myDefaultMethod() {
      // Code here
   }
}
```

Private:

The private access modifier restricts access to members only within the same class.

Members marked as private are not visible to other classes or subclasses.

Syntax

```
public class MyClass {
   // Private access modifier
   private int myPrivateVariable;
   private void myPrivateMethod() {
      // Code here
   }
}
```

Protected:

The protected access modifier allows access within the same package and by subclasses, regardless of their package.

Members with protected access are not visible outside the package unless accessed through inheritance.

Syntax

```
public class MyClass {
```

```
    // Protected access modifier
    protected int myProtectedVariable;
    protected void myProtectedMethod() {
        // Code here
    }
}
```

Public:

The public access modifier provides the widest accessibility, making members visible to any class, whether in the same package or a different one.

Syntax

```
public class MyClass {
    // Public access modifier
    public int myPublicVariable;
    public void myPublicMethod() {
        // Code here
    }
}
```

These access modifiers offer a range of options for developers to tailor the visibility and accessibility of their code, contributing to the overall design, security, and encapsulation of Java programs.

Note: Understanding and applying access control is crucial for creating maintainable, secure, and modular Java code. It allows developers to manage the visibility of their code components effectively.

Summary

- Java was developed by Sun Microsystems initially to offer solutions for household appliances. But, finally, it evolved as a fully functional programming language.
- Java has many features such as platform independence, simplicity, object-oriented capability, portability, and so on that differentiates it from other programming languages and makes it important.
- The main method has a specific signature: public static void main(String[] args).
- The main method serves as the entry point for Java applications.
- Access control is fundamental to encapsulation, where implementation details are hidden, and access is provided through well-defined interfaces.
- The Java uses access modifiers to control the visibility and accessibility of classes, methods, and variables.main method takes a single parameter, an array of strings (String[] args).
- Protected members are not visible outside the package unless accessed through inheritance.
- A class defines the properties (attributes) and behaviors (methods) that objects of the class will have.
- Polymorphism allows objects to be treated as instances of their parent class.

- Objects have instance members (attributes and methods specific to an instance) and class members (shared by all instances of a class).

Keywords

***class*:** Java class keyword is used to declare a class.

***float*:** Java float keyword is used to declare a variable that can hold a 32-bit floating-point number.

Object: Object means a real-world entity such as a pen, chair, table, computer, watch, etc.

Instance Variable: A variable that is relevant to a single instance (an object belonging to a class is an instance of that class) of a class.

OOP: Object-Oriented Programming.

Protected: It allows access within the same package and by subclasses, regardless of their package.

public: The public keyword is an access modifier that indicates that the main method can be accessed from outside the class.

static: The static keyword is used to declare the main method as a class-level method. It allows the method to be called without creating an instance of the class.

void: The void keyword is the return type of the main method, indicating that it does not return any value.

Encapsulation: Encapsulation is a fundamental principle of object-oriented programming (OOP) that involves bundling data (attributes) and methods that operate on the data within a single unit (class).

Self Assessment

1. What is the access modifier for the main method in Java?

A. private

B. protected

C. public

D. static

2. Why is the main method in Java declared as static?

A. To allow multiple instances of the main class

B. To enable method overloading

C. To allow the main method to be invoked without creating an instance of the class

D. To make the method thread-safe

3. What is the return type of the main method in Java?

A. int

B. void

C. String

D. Boolean

4. Which keyword is used to indicate command line arguments in the main method signature?

A. arguments

B. params

C. args

D. parameters

5. Which of the following is a feature of Java?
A. Multiple Inheritance
B. Pointers
C. Operator Overloading
D. Garbage Collection

6. What is the significance of the "platform-independent" feature in Java?
A. Java programs can run on any operating system without modification.
B. Java programs can run faster than programs written in other languages.
C. Java programs can only be executed on Windows.
D. Java programs require a specific version of the Java Virtual Machine (JVM).

7. Which feature allows a class to inherit members from multiple classes in Java?
A. Encapsulation
B. Polymorphism
C. Inheritance
D. Abstraction

8. What does the term "Object-Oriented" mean in the context of Java?
A. It refers to the ability to create standalone programs.
B. It emphasizes the use of pointers for memory management.
C. It focuses on organizing code into objects with attributes and behaviors.
D. It implies the exclusive use of procedural programming concepts.

9. What is a class in Java?
A. A template for creating objects
B. A built-in data type
C. A method for storing variables
D. An array of integers

10. Which keyword is used to create an object of a class in Java?
A. new
B. class
C. this
D. object

11. What is encapsulation in the context of Java classes?
A. It refers to hiding the implementation details of a class.
B. It involves creating multiple instances of a class.
C. It signifies the process of method overloading.
D. It is a way to declare variables inside a class.

12. In Java, what is the purpose of the "this" keyword in a class?
A. To create a new instance of the class
B. To refer to the current instance of the class

C. To access a static method
D. To declare a new class variable

13. In a Java program, if a class is declared with the public modifier, what is the scope of its accessibility?
A. Accessible only within the same class
B. Accessible within the same package
C. Accessible from any class in any package
D. Not accessible from any class

14. Inheritance in Java allows a class to:
A. Hide the implementation details
B. Create multiple instances
C. Inherit properties and behaviors from another class
D. Define a new class within an existing class

15. What is polymorphism in Java?
A. The ability of a class to have multiple constructors
B. The ability of a method to have multiple names
C. The ability of a class to inherit properties from multiple classes
D. The ability of an object to take on multiple forms, such as through method overloading or overriding

Answers for Self Assessment

1.	C	2.	C	3.	B	4.	C	5.	D
6.	A	7.	C	8.	C	9.	A	10.	A
11.	A	12.	B	13.	C	14.	C	15.	D

Review Questions

1. Analyze different features of Java, which has made Java an important programming language.
2. "To run a program in Java, the user can use a command prompt." Discuss.
3. Explain the concept of a class in Java and its role in object-oriented programming.
4. Describe the process of creating and using objects in Java. Give an appropriate example.
5. What is the primary purpose of the main method in Java? How are command-line arguments passed to the main method, and how can they be accessed within the program?
6. Explain the concept of access control in Java and discuss its importance in object-oriented programming.
7. Discuss the main access modifiers in Java by giving the example of each modifier.

Further Readings

Balagurusamy E. Programming with Java_A Primer 3e. New Delhi

Schildt. H. Java 2 The Complete Reference, 5th ed. New York: McGraw-Hill/Osborne.

King, K. N. (2000). Java programming: from the beginning. WW Norton & Co., Inc.

Sanchez, J., & Canton, M. P. (2002). Java programming for engineers. CRC Press.

Web Links

www.roseindia.net/java/java-introduction/java-features.shtml

http://java.sun.com/j2se/press/bama.html

http://docstore.mik.ua/orelly/java/javanut/ch03_01.htm

https://www.javatpoint.com/java-basics

https://www.geeksforgeeks.org/java-programming-basics/

Unit 02: Methods

CONTENTS

Objectives

After this unit you will be able to:

- Learn the concept of fields , methods and method declaration
- Implement the definition of fields and methods
- Understand and implement the different types of variables
- Know the different types of control structure

Introduction

a method is a block of code that performs a specific task or operation. Methods are the building blocks of Java programs and are used to define the behavior of objects, to perform actions, or to return values. Here's an introduction to methods in Java:

Anatomy of a Method

A method in Java has the following components:

Method Signature: This includes the method's name and the parameters it accepts (if any).

 Example:

public void methodName(int param1, String param2)

Return Type: This specifies the type of value that the method returns (if any).

 Example:

void (no return value), int, String, boolean, etc.

Access Modifier: This determines the visibility of the method.

 Example

public, private, protected, or package-private.

Method Body: This is enclosed within curly braces {} and contains the statements or instructions that define what the method does.

 Example:

```
public class MyClass {
   // Method with no parameters and no return value (void)
   public void greet() {
      System.out.println("Hello, World!");
   }

   // Method with parameters and return value
   public int add(int a, int b) {
      return a + b;
   }
   // Method with parameters and no return value (void)
   private void printDetails(String name, int age) {
      System.out.println("Name: " + name);
      System.out.println("Age: " + age);
   }

   // Main method - entry point of the program
   public static void main(String[] args) {
      MyClass obj = new MyClass();
      obj.greet(); // Calling the greet() method
      int sum = obj.add(5, 3); // Calling the add() method
      System.out.println("Sum: " + sum);
      obj.printDetails("John", 30); // Calling the printDetails() method
   }
}
```

2.1 Defining Fields And Methods

Defining fields and methods is fundamental to creating classes, which are the building blocks of object-oriented programming. Here's how you define fields (variables) and methods (functions) in Java:

Fields (Variables):

Fields represent the state of an object. They store data associated with the object. Fields can be of various data types such as int, double, String, etc.

 Example

```
public class MyClass {
   // Fields
   private int age;
   private String name;

   // Constructor
   public MyClass(int age, String name) {
      this.age = age;
      this.name = name;
   }
      // Methods
   public int getAge() {
      return age;
   }
      public String getName() {
      return name;
   }

   public void setAge(int age) {
      this.age = age;
   }

   public void setName(String name) {
      this.name = name;
   }
}
```

Methods (Functions):

Methods define the behavior of an object. They can perform operations, manipulate data, or provide information about the object's state.

 Example

```
class tt
{
int price;
String color;
 void getdata(int p,String c)
{
price=p;
color=c;
```

```
}
void display()
{
System.out.println("Price= "+price);
System.out.println("Color= "+color);
}
}
class methodex
{
public static void main(String args[])
{
tt t=new tt();
t.getdata(230,"Red");
t.display();
}
}
```

Output

```
PS D:\392\Programs> javac methodex.java
PS D:\392\Programs> java methodex
Price= 230
Color= Red
```

2.2 Method Arguments And Return Values

//With argument without return type

 Example

```
class st
{
 int id;
 String name;
void getid(int r,String n)
{
id=r;
name=n;
}
void show()
{
System.out.println("Roll "+id);
System.out.println("Name= "+name);
}
```

```
}
class test
{
public static void main(String args[])
{
st a=new st();
a.getid(12,"Seerat");
a.show();
}
}
```

Output

```
PS D:\392\Programs> javac test.java
PS D:\392\Programs> java test
Roll 12
Name= Seerat
```

2.3 Declaring

Declaring a method involves specifying its name, return type, parameters (if any), and access level. Here's a basic syntax for declaring a method:

```
accessSpecifier returnType methodName(parameterType1 parameter1, parameterType2 parameter2, ...) {
   // Method body
}
```

where:

Access Specifier: Specifies the visibility of the method. It can be public, protected, private, or package-private (no explicit modifier).

Return Type: Specifies the type of value that the method returns. If the method does not return any value, you can use the void keyword.

Method Name: The name of the method, which is used to call it.

Parameters: Input values that are passed to the method. Parameters are optional. If a method doesn't take any parameters, you'll see empty parentheses ().

Method Body: Contains the statements that define what the method does.

Here's a simple example:

 Example

```
public class MyClass {
   // Method declaration
   public int add(int a, int b) {
      return a + b; // Method body
   }
}
```

In this example:

Access Specifier: public - This method can be accessed from any other class.

Return Type: int - This method returns an integer value.

Method Name: add - Name of the method.

Parameters: int a and int b - Two integer parameters are passed to the method.

Method Body: return a + b; - The method adds the two integers and returns the result.

Remember, method declarations define the signature and behavior of a method, while method calls actually execute the code within the method body.

2.4 Instantiating and Initializing Objects

In Java, you instantiate and initialize objects using the new keyword along with a constructor. Here's a step-by-step guide:

Declare a Class: First, you need to have a class that defines the blueprint for your objects. Here's an example of a simple class:

 Example

```
public class MyClass {
   // Fields
   private int myField;
      // Constructor
   public MyClass(int initialValue) {
      this.myField = initialValue;
   }
      // Methods
   public int getMyField() {
      return myField;
   }
}
```

Instantiate an Object: Once you have a class, you can create objects of that class using the new keyword followed by a call to the class constructor.

 Example

```
MyClass obj = new MyClass(10);
```

In this line, MyClass is the class name, obj is the name of the variable that holds the reference to the newly created object, new is the keyword that allocates memory for the object, and (10) is a call to the constructor of MyClass with an initial value of 10.

Initialize Object Fields: If your class has any fields that need initialization, you can do it either directly within the constructor or through setter methods.

 Example

```
public MyClass(int initialValue) {
   this.myField = initialValue;
}
```

In this constructor, the field myField is initialized with the value passed to the constructor.

Access Object Members: Once you've instantiated an object, you can access its fields and methods using the dot (.) operator.

 Example

```
int value = obj.getMyField();

System.out.println(value); // Output: 10
```

In this example, getMyField() is a method of the MyClass object obj, which returns the value of the myField variable. This is the basic process of instantiating and initializing objects in Java. It's fundamental to object-oriented programming and allows you to create instances of classes to work with data and behavior defined within those classes.

2.5 Variables And Its Types

variables are containers for storing data values. They hold different types of data, such as numbers, characters, or objects. Java supports various types of variables, including primitive data types and reference types. Here's an overview of the types of variables in Java:

Primitive Data Types: Primitive data types are the most basic data types in Java. They hold single values and are predefined by the language. Java has eight primitive data types:

- byte: 8-bit signed integer. Example: byte age = 30;
- short: 16-bit signed integer. Example: short distance = 1000;
- int: 32-bit signed integer. Example: int count = 10000;
- long: 64-bit signed integer. Example: long population = 7000000000L;
- float: 32-bit floating point. Example: float price = 24.99f;
- double: 64-bit floating point. Example: double pi = 3.14159;
- char: 16-bit Unicode character. Example: char grade = 'A';
- boolean: Represents true or false. Example: boolean isTrue = true;

Reference Types: Reference types are used to store references (memory addresses) to objects. They don't store the actual data but rather a pointer to where the data is stored in memory. Reference types include:

Class Types: Objects of user-defined classes.

 Example: MyClass obj = new MyClass();

Array Types: Arrays, which are collections of elements of the same type.

 Example: int[] numbers = {1, 2, 3, 4, 5};

Interface Types: Interfaces, which define a set of methods.

 Example: Runnable task = () -> System.out.println("Running task");

Enum Types: Enumerated types, which represent a fixed set of constants.

 Example: Season season = Season.SUMMER;

Variables in Java can also be classified based on their scope:

Local Variables: Declared inside a method, constructor, or block. They are accessible only within the scope where they are declared.

 Example

```
void myMethod() {
   int x = 10; // Local variable
}
```

Instance Variables (Non-Static Fields): Variables declared within a class but outside any method. Each instance of the class (object) has its own copy of these variables.

 Example

```
class MyClass {
   int y; // Instance variable
}
```

Class Variables (Static Fields): Variables declared with the static keyword inside a class. They are shared among all instances of the class.

 Example

```
class MyClass {
   static int z; // Class variable
}
```

2.6 Control Flow Constructs

Control flow constructs in Java are the mechanisms used to control the flow of execution in a program. They determine the order in which statements are executed based on conditions or loops. Here are the main control flow constructs in Java:

Conditional Statements:

if: Executes a block of code if a specified condition is true.

Syntax

```
if (condition) {
   // Code to be executed if condition is true
}
```

if-else: Executes one block of code if the condition is true and another block if it's false.

Syntax

```
if (condition) {
   // Code to be executed if condition is true
} else {
   // Code to be executed if condition is false
}
```

 **Example**

```
class iels
{
public static void main(String args[])
{
int n=-10;
if(n>0)
System.out.println(n+" is positive");
else
System.out.println(n+" is negative");
```

```
}
}
```

Output

```
PS D:\392\Programs> javac iels.java
PS D:\392\Programs> java iels
-10 is negative
```

else-if: Allows checking multiple conditions in sequence.

Syntax

```
if (condition1) {
   // Code to be executed if condition1 is true
} else if (condition2) {
   // Code to be executed if condition2 is true
} else {
   // Code to be executed if none of the conditions are true
}
```

If-else if ladder

```
class ladder
{
public static void main(String args[])
{
int a=14;
int b=12;
int c=15;
if(a>b)
if(a>c)

System.out.println(" A is the largest");
else
System.out.println("No harm");
else
System.out.println("try some other input");
}
}
```

Output

```
PS D:\392\Programs> javac ladder.java
PS D:\392\Programs> java ladder
No harm
```

Switch Statement:

Executes one block of code from multiple alternatives based on the value of an expression.

Syntax

```
switch (expression) {
   case value1:
      // Code to be executed if expression equals value1
      break;
   case value2:
      // Code to be executed if expression equals value2
      break;
   // Additional cases...
   default:
      // Code to be executed if expression doesn't match any case
}
```

Loops:

for Loop: Executes a block of code a specified number of times.

Syntax

```
for (initialization; condition; update) {
   // Code to be executed repeatedly
}
```

 **Example**

```
class forexample
{
public static void main(String args[])
{
int i;
int s=0;
for(i=1;i<=10;i++)
{
s=s+i;
}
System.out.println("Sum= "+s);
}
}
```

Output

```
PS D:\392\Programs> javac forexample.java
PS D:\392\Programs> java forexample
Sum= 55
```

while Loop: Executes a block of code as long as a specified condition is true.

Syntax

```
while (condition) {
```

```
    // Code to be executed repeatedly
}
```

do-while Loop: Similar to the while loop but ensures the code block is executed at least once before checking the condition.

Syntax

```
do {
    // Code to be executed repeatedly
} while (condition);
```

Branching Statements:

break: Terminates the innermost loop or switch statement.

continue: Skips the rest of the loop's current iteration and proceeds to the next iteration.

return: Exits from the current method, optionally returning a value.

These control flow constructs provide the necessary tools to create algorithms that can make decisions, iterate over collections, and handle complex logic in Java programs.

Summary

- Methods in Java are encapsulated blocks of code performing specific tasks.
- They consist of a return type, method name, optional parameters, and a method body.
- Access modifiers control the visibility of methods.
- Methods promote code reusability and organization.
- They can return values of various types or nothing (void).
- Method overloading allows multiple methods with the same name but different parameter lists.
- Flow control in Java manages the sequence of execution within a program.
- Conditional statements, such as if, else, else if, and switch, allow for executing code based on specified conditions.
- Loops, including for, while, and do-while, enable repetitive execution of code blocks until certain conditions are met.
- Branching statements, like break, continue, and return, alter the flow of control within loops, switches, and methods.
- Control flow constructs are crucial for implementing decision-making logic, iteration, and handling exceptions in Java programs.

Keywords

Access Modifiers: Control the visibility of methods (public, protected, private, or default).

Return Type: Specifies the type of value returned by the method. Use void if the method doesn't return anything.

Method Name: Identifier used to invoke the method. Should follow Java naming conventions.

Parameters: Input values passed to the method. Optional, and a method may have none, one, or multiple parameters.

Method Body: Contains statements defining what the method does. Enclosed within curly braces {}.

Return Statement: Exits the method and optionally returns a value.

Overloading: Allows multiple methods with the same name but different parameter lists.

for: Executes a block of code a specified number of times.

while: Executes a block of code as long as a specified condition is true.

do-while: Similar to while loop but ensures the code block is executed at least once before checking the condition.

Self Assessment

1. In Java, which keyword is used to define a method that does not return any value?

A. void
B. return
C. int
D. double

2. What is the purpose of the "break" statement in a loop?

A. To skip the current iteration and continue with the next iteration
B. To exit the loop immediately
C. To restart the loop from the beginning
D. To pause the loop temporarily

3. Which control structure is used to execute a block of code repeatedly as long as a condition is true?

A. If-else
B. For loop
C. Switch case
D. While loop

4. What does the "continue" statement do in a loop?

A. Exits the loop immediately
B. Skips the remaining code in the loop and continues with the next iteration
C. Restarts the loop from the beginning
D. Pauses the loop temporarily

5. In Java, how is a method declared?

A. Using the keyword "function"
B. Using the keyword "method"
C. Using the keyword "define"
D. Using the keyword "void"

6. Which control structure is used to make decisions based on multiple conditions in Java?

A. If-else
B. For loop
C. Switch case

D. While loop

7. What is the primary purpose of the "else" clause in an if-else statement in Java?

A. To execute a block of code if the condition is true
B. To execute a block of code if none of the preceding conditions are true
C. To terminate the program
D. To define a variable

8. In Java, how is a function defined?

A. Using the keyword "method"
B. Using the keyword "define"
C. Using the keyword "function"
D. Using the keyword "def"

9. Which keyword in Java is used to indicate that a method can be accessed and called by code in other classes?

A. public
B. private
C. protected
D. static

10. What is the output of the following code snippet written in Java?

java

Copy code

```
int x = 5;
if (x > 10) {
   System.out.println("x is greater than 10");
} else if (x > 7) {
   System.out.println("x is greater than 7");
} else {
   System.out.println("x is less than or equal to 7");
}
```

A. x is greater than 10
B. x is greater than 7
C. x is less than or equal to 7
D. The code will produce an error

11. What is the correct syntax for declaring a method named "calculateSum" that takes two integer parameters and returns an integer in Java?

A. int calculateSum(int a, int b) { }
B. void calculateSum(int a, int b) { }
C. calculateSum(int a, int b) { }
D. int calculateSum(a, b) { }

12. Which of the following is NOT a valid type of loop in Java?

A. For loop
B. While loop
C. If-else loop
D. Do-while loop

13. What is the purpose of the "return" statement in a method in Java?

A. To terminate the program
B. To skip the current iteration of a loop
C. To exit the method and return a value to the caller
D. To restart the method from the beginning

14. In Java, how are method parameters passed?

A. By value
B. By reference
C. By copying the entire memory
D. By creating a new instance of the parameter

15. What is the purpose of the "this" keyword in Java?

A. To refer to the current object instance
B. To create a new object instance
C. To refer to the superclass of the current class
D. To declare a static method

Answers for Self Assessment

1.	A	2.	B	3.	D	4.	B	5.	D
6.	C	7.	B	8.	A	9.	A	10.	C
11.	A	12.	C	13.	C	14.	A	15.	A

Review Questions

1. Explain the purpose and significance of methods in Java programming. Discuss how methods contribute to code organization, reusability, and modularity.
2. Compare and contrast the if-else statement, switch statement, and ternary operator (? :) in Java. Discuss their similarities, differences, and best use cases.
3. Describe the concept of method overloading in Java. Provide examples to demonstrate how method overloading works and discuss its benefits in software development.
4. Discuss the differences between static methods and instance methods in Java. Explain when to use each type of method and provide examples to illustrate their usage.

5. Explain the importance of access modifiers (e.g., public, private, protected, default) in Java methods. Discuss the visibility and accessibility of methods based on different access modifiers.
6. Discuss the role of parameters and return types in Java methods. Explain how parameters are passed to methods and how return values are utilized in method invocations.
7. Describe the purpose and functionality of each of the following loop constructs in Java: for loop, while loop, and do-while loop. Provide examples to demonstrate their usage and discuss when each loop type is appropriate.

Further Readings

Deitel, P. J., & Deitel, H. M. (2009). Java for programmers. Pearson education.

Kendal, S. (2009). Object oriented programming using Java. Bookboon.

Dale, N. B., Joyce, D. T., & Weems, C. (2002). Object-oriented data structures using Java. Jones & Bartlett Learning.

Dale, N. B., Joyce, D. T., & Weems, C. (2002). Object-oriented data structures using Java. Jones & Bartlett Learning.

Friesen, J. (2012). Beginning Java 7. Apress.

Fain, Y. (2011). Java programming 24-hour trainer. John Wiley & Sons

Web Links

https://www.javatpoint.com/method-in-java

https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html

https://www.simplilearn.com/tutorials/java-tutorial/methods-in-java

https://www.freecodecamp.org/news/java-methods/

https://www.baeldung.com/java-control-structures

https://javagoal.com/control-structures-in-java/

.

Unit 03: Encapsulation & Polymorphism

CONTENTS

Objectives

After this unit you will be able to:

- Learn the concept of encapsulation, polymorphism, and operator overloading
- Implement encapsulation, polymorphism, and operator overloading

Introduction

Encapsulation is a fundamental concept in Java and object-oriented programming (OOP) that revolves around bundling data and methods that operate on that data within a single unit known as a class. The primary goal of encapsulation is to hide the internal implementation details of a class and expose a well-defined public interface. This concept helps in achieving data security, flexibility, and code maintainability. In Java, encapsulation is implemented by declaring the variables of a class as private, restricting direct access from outside the class. Access to these private variables is provided through public methods, commonly known as getter and setter methods. Getter methods allow the retrieval of values, and setter methods enable the modification of values while enforcing any necessary validation rules.

Polymorphism, another key concept in Java and OOP, enables a single entity, such as a method or object, to take on multiple forms. This flexibility allows developers to write more versatile and adaptable code. Polymorphism is achieved in Java through two main types: compile-time polymorphism and runtime polymorphism. Polymorphism enhances code flexibility, readability, and maintainability by allowing developers to write generic code that can work with a variety of objects. It aligns with the principles of abstraction, encapsulation, and inheritance in OOP, contributing to the creation of modular and extensible software systems. In Java, polymorphism is a powerful mechanism that enables developers to build scalable and adaptable solutions.

3.1 Encapsulation

Encapsulation in Java stands as a fundamental principle of object-oriented programming (OOP), emphasizing the consolidation of data and associated methods within a single unit known as a class. This concept shields the internal implementation details of a class, exposing only a public interface for external interactions. In Java, encapsulation is implemented by declaring class instance variables as private, restricting direct access from outside the class. External access to these variables is facilitated through public methods called getters and setters. Getters retrieve the values of instance variables, while setters modify them. This design allows the class to enforce specific data validation rules, ensuring the integrity of its internal state. By utilizing getters and setters, a class can carefully control external interactions and maintain a consistent and secure internal structure.

Encapsulation serves as the encapsulation of data within a unified entity, creating a connection between code and the data it manages. It acts as a protective barrier, preventing external code from directly accessing the encapsulated data. In a technical sense, encapsulation involves concealing the variables or data of a class, allowing access solely through member functions within that class. This data-hiding practice is realized by marking class members or methods as private. The class is then presented to the external world using abstraction, withholding implementation details. Encapsulation, therefore, emerges as a blend of data hiding and abstraction. It is achieved by designating all class variables as private and providing public methods (getters and setters) to manipulate these variables. This methodology enhances security and control, defining a clear interface for external interactions while concealing the inner workings of the class.

 Example

```
class Encapsulate {
        private String Name;
        private int Roll;
        private int Age;
        public int getAge() { return Age; }



        public String getName() { return Name; }
        public int getRoll() { return Roll; }
public void setAge(int nAge) { Age = nAge; }
        public void setName(String nName)
        {
                Name = nName;
        }
        public void setRoll(int nRoll) { Roll = nRoll; }
}
public class TestEncapsulation {
        public static void main(String[] args)
        {
                Encapsulate obj = new Encapsulate();
                // setting values of the variables
                obj.setName("Seerat");
                obj.setAge(12);
```

```
        obj.setRoll(15);
        // Displaying values of the variables
        System.out.println("Name: " + obj.getName());
        System.out.println("Age: " + obj.getAge());
        System.out.println("Roll: " + obj.getRoll());
            }
}
```

Output

```
PS D:\392\Programs> javac TestEncapsulation.java
PS D:\392\Programs> java TestEncapsulation
Name: Seerat
Age: 12
Roll: 15
```

3.2 Advantages of Encapsulation

Data Hiding: Encapsulation is a mechanism to restrict access to data members, concealing the internal implementation details. Users are shielded from the intricacies of how values are stored in variables; they only interact with setter methods, initializing variables with provided values.

Increased Flexibility: The flexibility of encapsulation is demonstrated in the ability to customize variable access. By omitting setter methods like setName(), setAge(), etc., variables can be made read-only. Conversely, by excluding get methods such as getName(), getAge(), etc., variables become write-only.

Reusability: Encapsulation enhances code reusability, making it adaptable to new requirements. The encapsulated class can be easily integrated into different contexts without compromising its internal structure.

Testing Code is Easy: Encapsulated code facilitates unit testing. With clear boundaries and encapsulated functionality, testing becomes more straightforward, allowing for efficient identification and resolution of issues.

Freedom for Programmers: Encapsulation empowers programmers by providing the freedom to implement system details. While adhering to the abstract interface visible to outsiders, programmers have the flexibility to modify the internal workings of the system as needed. This balance between freedom and a defined interface contributes to a robust and adaptable codebase.

3.3 Disadvantages of Encapsulation in Java

Potential for Increased Complexity: Improper usage of encapsulation can introduce complexity to the codebase, making it harder to comprehend and maintain. Overuse or misapplication of encapsulation principles may lead to intricate design patterns that hinder rather than enhance the overall system.

Reduced Transparency in System Understanding: Excessive encapsulation may obscure the inner workings of the system, making it more challenging for developers to understand the code. This lack of transparency can impede collaboration and troubleshooting efforts, particularly for those unfamiliar with the encapsulated structure.

Limitation on Implementation Flexibility: While encapsulation promotes a clear separation between interface and implementation, it may impose constraints on the flexibility of system implementation. Striking the right balance is crucial to ensure that encapsulation enhances code organization without compromising adaptability.

3.4 Polymorphism

Polymorphism in Java is a fundamental concept that enables the execution of a single action in diverse ways. The term "polymorphism" originates from the Greek words "poly," meaning many, and "morphs," meaning forms. Essentially, polymorphism embodies the idea of having many forms.

Within the realm of Object-Oriented Programming, polymorphism stands out as a crucial feature. It allows for the flexibility of performing a specific operation through various implementations, emphasizing adaptability and versatility in the code structure.

Java Polymorphism can be categorized into two primary types:

Compile-time Polymorphism: Also referred to as static polymorphism, compile-time polymorphism is accomplished through function overloading or operator overloading.

Notably, Java does not support operator overloading.

Method Overloading: Method overloading occurs when multiple functions share the same name but differ in their parameters.

Overloading can involve changes in the number and/or types of arguments, allowing for a more flexible and expressive usage of functions.

 Example

```
class Area
{
public int area(int x)
{
return(x*x);
}
public int area(int x,int y)
{
return(x*y);
}
}
class Exa
{
public static void main(String args[])
{
Area a=new Area();
System.out.println("Area of Square= "+a.area(5));
System.out.println("Area of Rectangle= "+a.area(10,10));
}
}
```

Output

```
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> javac Exa.java
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> java Exa
Area of Square= 25
```

Area of Rectangle= 100

Runtime Polymorphism: Runtime polymorphism, also known as Dynamic Method Dispatch, is a process where the resolution of a call to an overridden method takes place at runtime instead of compile-time. In this mechanism, the overridden method is invoked through a reference variable of a superclass, and the determination of which method to call is based on the actual object being referenced. Before reaching into Runtime Polymorphism, it's crucial to grasp the concept of upcasting.

Upcasting: When a reference variable of the parent class refers to an object of the child class, it is termed as upcasting.

Runtime Polymorphism in Java: Dynamic Method Dispatch, or Runtime Polymorphism, is achieved through Method Overriding. Method overriding occurs when a derived class provides its own definition for a member function of the base class, effectively replacing or "overriding" the original definition. When an object of a child class is created, the method within the child class is invoked. This is because the method in the parent class has been overridden by the child class, granting the overridden method in the child class higher priority. Consequently, the body of the method inside the child class is executed during the runtime.


Example

```
//Method overriding without paramater
class riding
{
void display()
{
System.out.println("You are in the riding class");
}
}
class swiming extends riding
{
void display()
{
System.out.println("You are in swiming class");
}
}
class cyclying extends riding
{
void display()
{
System.out.println("You are learning how to drive cycle");
}
}
class Exa1
{
public static void main(String args[])
{
```

```
riding r=new riding();
r.display();
r=new swiming();
r.display();
r=new cyclying();
r.display();
}
}
```

Output

```
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> javac Exa1.java
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> java Exa1
You are in the riding class
You are in swiming class
You are learning how to drive cycle
```

 Example

```
//Method overriding without paramater
class rectangle
{
void area(int x,int y)
{
System.out.println("Area of rectangle= "+(x*y));
}
}
class addition extends rectangle
{
void area(int x,int y)
{
System.out.println("Addition of two numbers= "+(x+y));
}
}
class Exa2
{
public static void main(String args[])
{
rectangle r=new rectangle();
r.area(4,6);
r=new addition();
r.area(34,10);
}
```

```
}
```

Output

```
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> javac Exa2.java
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> java Exa2
Area of rectangle= 24
Addition of two numbers= 44
```

3.5 Java Runtime Polymorphism with Data Member

A method is overridden, not the data members, so runtime polymorphism can't be achieved by data members.

In the example given below, both the classes have a data member roll. We are accessing the data member by the reference variable of Parent class which refers to the subclass object. Since we are accessing the data member which is not overridden, hence it will access the data member of the Parent class always.


Example

```
class student
{
int roll=10;
}
class stu extends student
{
int roll=20;
}
class d
{
public static void main(String args[])
{
student s=new student();
System.out.println("Parent class "+s.roll);
s=new stu();
System.out.println("Child class "+s.roll);
}
}
```

Output

```
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> javac d.java
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> java d
Parent class 10
Child class 10
```

3.6 Method Overloading

Method Overloading facilitates the definition of multiple methods with the same name but distinct signatures. The signature differences may involve variations in the number or types of input parameters, or a combination of both.

In Java, Method Overloading is alternatively termed Compile-time Polymorphism, Static Polymorphism, or Early Binding. When employing Method Overloading, the argument from the child class takes precedence over the parent argument, ensuring that the overridden method in the child class holds the highest priority.

Various ways exist to implement Method Overloading in Java, providing flexibility in method definition. These include:

Changing the Number of Parameters: Overloading can be achieved by altering the number of parameters in a method. Multiple methods with the same name but different parameter counts can coexist. Method overloading is realized by adjusting the number of parameters passed to distinct methods.

```
// Method with two parameters
public int add(int a, int b) {
   // Method implementation for two parameters
}

// Method with three parameters, overloading the previous method
public int add(int a, int b, int c) {
   // Method implementation for three parameters
}
```

In this example, the add method is overloaded by creating a version with two parameters and another version with three parameters. The method name remains the same, but the number of parameters varies, demonstrating method overloading in Java.

Example

```
class Addition {
                public int add(int a, int b)
        {
                int prod = a + b;
                return prod;
        }

        public int add(int a, int b, int c)
        {
                int prod = a + b +c;
                return prod;
        }
}

// Class 2
```

```
// Main class
class MO {
	// Main driver method
	public static void main(String[] args)
	{
		// Creating object of above class inside main()
		// method
		Addition ob = new Addition();

		// Calling method to Multiply 2 numbers
		int prod1 = ob.add(1, 2);

		// Printing Product of 2 numbers
		System.out.println(
			"Sum of the two integer value :" + prod1);

		// Calling method to multiply 3 numbers
		int prod2 = ob.add(1, 2, 3);

		// Printing product of 3 numbers
		System.out.println(
			"Sum of the three integer value :" + prod2);
	}
}
```

Output

```
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> javac MO.java
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> java MO
Sum of the two integer value :3
Sum of the three integer value :6
```

Changing Data Types of the Arguments: Method Overloading allows the use of the same method name with different data types for its parameters. This enables developers to create distinct implementations based on the type of input data. Methods are deemed overloaded when they share the same name but differ in parameter types.

```
int prod(int a, int b, int c) {
  // Method implementation for integer parameters
}

// Method with three double parameters, overloading the previous method
public double prod(double a, double b, double c) {
  // Method implementation for double parameters
}
```

In this illustration, the prod method is overloaded by offering versions for both integer and double parameter types. Despite having the same name, the methods serve distinct data types, showcasing method overloading in Java.


Example

```
class Addition {
        public int add(int a, int b)
    {
        int prod = a + b;
        return prod;
    }

    public double add(double a, double b)
    {
        double prod = a + b;
        return prod;
    }
}
class MO {
    // Main driver method
    public static void main(String[] args)
    {

        Addition ob = new Addition();
        int prod1 = ob.add(1, 2);
        System.out.println( "Sum of the two integer value :" + prod1);
        double prod2 = ob.add(1.5,3.6);
        System.out.println( "Sum of the three integer value :" + prod2);
    }
}
```

Output

```
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> javac MO.java
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> java MO
Sum of the two integer value :3
Sum of the three integer value :5.1
```

Changing the Order of the Parameters of Methods: The order of parameters in a method can be modified to achieve overloading. By rearranging the sequence of parameters, multiple methods with the same name but different parameter orders can be defined. Method overloading is also achievable by reshuffling the parameters among two or more methods with the same name. For instance, if method 1 has parameters (String name, int roll_no) and another method has parameters (int roll_no, String name), both bearing the same name, these methods are regarded as overloaded with distinct parameter sequences.

```
// Method with parameters (String name, int roll_no)
public void studentId(String name, int roll_no) {
   // Method implementation with the first parameter sequence
}

// Method with parameters (int roll_no, String name), overloading the previous method
public void studentId(int roll_no, String name) {
   // Method implementation with the second parameter sequence
}
```

In this scenario, the studentId method is overloaded by rearranging the order of parameters between two versions of the method, exemplifying method overloading in Java.

3.7 Benefits of Method Overloading

Enhanced Readability and Reusability: Method overloading contributes to the clarity and reuse of code. Having multiple methods with the same name, each catering to specific parameter variations, makes the program more readable and easier to comprehend.

Reduced Program Complexity: Method overloading aids in simplifying program structures by consolidating related functionalities under a single method name. This not only streamlines the codebase but also makes it more manageable.

Efficient Task Execution: Programmers can efficiently execute tasks using method overloading. By defining methods with the same name but different parameter sets, developers can invoke the desired functionality based on the specific requirements.

Flexible Access to Related Functions: Method overloading allows access to methods that perform similar functions with slight variations in arguments and types. This flexibility enhances the adaptability of the program to diverse scenarios.

Summary

- Encapsulation is a fundamental concept in Object-Oriented Programming (OOP) that involves bundling data and methods within a single unit, known as a class.
- Encapsulation enforces data hiding by making class variables private, restricting direct access.
- Encapsulation promotes data integrity by encapsulating and controlling access to class members.
- Encapsulation helps prevent unauthorized modification and ensures controlled interaction through designated methods.
- Polymorphism, a key concept in Object-Oriented Programming (OOP), allows a single operation to be performed in different ways, depending on the context.
- Compile-time Polymorphism (Static) is achieved through function overloading or operator overloading (not supported in Java).
- Runtime Polymorphism (Dynamic) is implemented through method overriding, where the call to an overridden method is resolved at runtime.
- Upcasting refers to assigning a reference variable of a superclass to an object of a subclass. Enables flexibility and dynamic method invocation.
- Method overloading is a feature in Java that allows the definition of multiple methods in the same class with the same name but different parameter lists.
- Overloading can occur by adjusting the count of parameters in a method.
- Methods with the same name may differ in the data types of their parameters.

- Overloading can also involve rearranging the order of parameters.

Keywords

Polymorphism: It is a key concept in Object-Oriented Programming (OOP), allows a single operation to be performed in different ways, depending on the context.

Function Overloading: Method names remain the same, but parameters differ in type, number, or both. Enhances code readability and provides flexibility in method usage.

Method Overriding: Occurs in a derived class where a method in the base class is redefined, providing a specific implementation. Resolves calls dynamically based on the object type.

Dynamic Method Dispatch: A mechanism where the determination of which overridden method to call happens at runtime based on the actual object being referenced.

Upcasting: Refers to assigning a reference variable of a superclass to an object of a subclass. Enables flexibility and dynamic method invocation.

Abstraction: It embraces the abstraction principle, offering a simplified and well-defined external interface while concealing the intricate details of the internal implementation.

Access Control: Access modifiers (public, private, protected) are utilized to control the visibility of class members, allowing fine-grained access control and adhering to the principle of least privilege.

Self Assessment

1. What is encapsulation in Java?

A. A type of loop structure
B. A mechanism for bundling data and methods within a class
C. An arithmetic operation
D. A sorting algorithm

2. How does encapsulation enhance security in Java?

A. By using complex encryption algorithms
B. By restricting access to class members through access modifiers
C. By implementing firewalls in the code
D. By using antivirus software

3. Which access modifier is commonly used for private variables in Java encapsulation?

A. public
B. protected
C. private
D. default (package-private)

4. What is the purpose of getter methods in encapsulation?

A. To set values for class variables
B. To perform complex calculations
C. To retrieve the values of private variables
D. To create new instances of a class

5. In encapsulation, what is the role of abstraction?

A. Making code more complex
B. Hiding internal implementation details while exposing a clear interface
C. Allowing direct access to private variables
D. Avoiding the use of access modifiers

6. What is polymorphism in Java?
A. A sorting algorithm
B. A type of loop structure
C. A mechanism to perform a single action in different ways
D. An encryption technique

7. Which type of polymorphism is achieved through method overriding?
A. Compile-time polymorphism
B. Runtime polymorphism
C. Static polymorphism
D. Early binding

8. What is the other term commonly used for compile-time polymorphism in Java?
A. Dynamic polymorphism
B. Late binding
C. Method overloading
D. Operator overloading

9. What is dynamic method dispatch in Java polymorphism?
A. A technique for dynamic method creation
B. A mechanism to change method signatures at runtime
C. The process of resolving overridden methods at runtime
D. A way to dispatch methods to different classes dynamically

10. Which keyword is used in Java to implement method overriding?
A. override
B. redefine
C. extends
D. @Override

11. What is method overloading in Java?
A. A technique for encrypting methods
B. A mechanism for defining multiple methods with the same name
C. A method for loading external libraries
D. A way to override existing methods

12. How can method overloading be achieved in Java?
A. By declaring all methods as private
B. By using the 'override' keyword
C. By defining methods with the same name but different parameter lists
D. By making all methods static

13. Which type of polymorphism is associated with method overloading?
A. Compile-time polymorphism
B. Runtime polymorphism
C. Dynamic polymorphism
D. Operator polymorphism

14. In method overloading, what must differ between the overloaded methods?

A. Method name
B. Return type
C. Parameter list
D. Access modifier

15. What is one advantage of method overloading in Java?
A. Increased code complexity
B. Reduced code readability
C. Improved code reusability
D. Limited flexibility in method calls

Answers for Self Assessment

1.	B	2.	B	3.	C	4.	C	5.	B
6.	C	7.	B	8.	C	9.	C	10.	D
11.	B	12.	C	13.	A	14.	C	15.	C

Review Questions

1. Explain the concept of encapsulation in Java. Why is it considered a fundamental principle in Object-Oriented Programming (OOP)?
2. Discuss the role of access modifiers in encapsulation. How do private, public, and protected modifiers contribute to controlling access to class members?
3. Illustrate how encapsulation is implemented through getter and setter methods in Java. Provide an example to demonstrate the use of encapsulation with these methods.
4. Explain the concept of polymorphism in Java. How does it contribute to the flexibility and adaptability of object-oriented programming?
5. Differentiate between compile-time polymorphism and runtime polymorphism in Java. Provide examples to illustrate each type.
6. Explain the concept of method overloading in Java with the help of appropriate example.
7. Discuss the advantages of method overloading in Java. How does it improve code readability, flexibility, and code maintenance?

Further Readings

Deitel, P. J., & Deitel, H. M. (2009). Java for programmers. Pearson education.

Kendal, S. (2009). Object oriented programming using Java. Bookboon.

Dale, N. B., Joyce, D. T., & Weems, C. (2002). Object-oriented data structures using Java. Jones & Bartlett Learning.

Dale, N. B., Joyce, D. T., & Weems, C. (2002). Object-oriented data structures using Java. Jones & Bartlett Learning.

Friesen, J. (2012). Beginning Java 7. Apress.

Fain, Y. (2011). Java programming 24-hour trainer. John Wiley & Sons.

Web Links

https://www.programiz.com/java-programming/encapsulation

https://blog.hubspot.com/website/encapsulation-java

https://www.mygreatlearning.com/blog/polymorphism-in-java/

https://www.geeksforgeeks.org/difference-between-method-overloading-and-method-overriding-in-java/

https://www.softwaretestinghelp.com/polymorphism-in-java/

Unit 04: Constructors

CONTENTS

Objectives

- Define what constructors are and their role in Java programming.
- Recognize the syntax for declaring constructors.
- Explain the difference between constructors and methods.

Introduction

In Java, a constructor is a special type of method that is called when an object is instantiated. Constructors have the same name as the class and do not have a return type, not even void. Constructors are used to initialize the object's state. If a class does not explicitly define any constructors, Java provides a default constructor with no parameters, which assigns default values to the members of the class according to their types.

4.1 Constructors in Java

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory. **Java constructors** are special types of methods that are used to initialize an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class or to perform any other start-up procedures required to create a fully formed object.

All classes have constructors, whether you define one or not because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your constructor, the default constructor is no longer used.

Java constructors are special types of methods that are used to initialize an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class or to perform any other start-up procedures required to create a fully formed object.

All classes have constructors, whether you define one or not because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your constructor, the default constructor is no longer used.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

4.2 Rules for creating Java Constructor

There are two rules defined for the constructor.

Constructor name must be the same as its class name

A Constructor must have no explicit return type

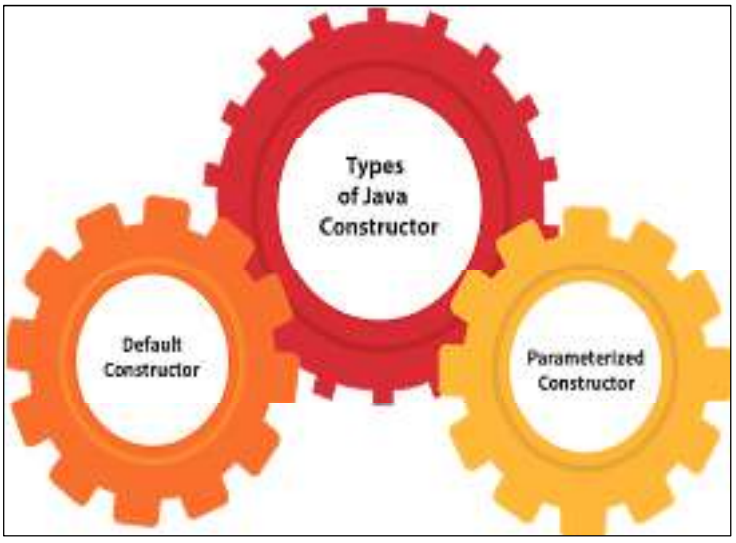
A Java constructor cannot be abstract, static, final, and synchronized

Note: We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

4.3 Types of Java Constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

1. <class_name>(){}

Example of default constructor

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
//Java Program to create and call a default constructor
class Bike1{
//creating a default constructor
Bike1(){System.out.println("Bike is created");}
//main method
public static void main(String args[]){
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

Output:

Bike is created

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.

What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Example of default constructor that displays the default values

```
//Let us see another example of default constructor
//which displays the default values
class Student3{
int id;
String name;
//method to display the value of id and name
```

```
void display(){System.out.println(id+" "+name);}

public static void main(String args[]){
//creating objects
Student3 s1=new Student3();
Student3 s2=new Student3();
//displaying values of the object
s1.display();
s2.display();
}
}
```

Output:

0 null

0 null

Explanation: In the above class,you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

4.4 Java Parameterized Constructor

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

Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student4{
  int id;
  String name;
  //creating a parameterized constructor
  Student4(int i,String n){
  id = i;
  name = n;
  }
  //method to display the values
  void display(){System.out.println(id+" "+name);}

  public static void main(String args[]){
  //creating objects and passing values
  Student4 s1 = new Student4(111,"Karan");
  Student4 s2 = new Student4(222,"Aryan");
```

```
    //calling method to display the values of object
    s1.display();
    s2.display();
   }
1.  }
```

Output:

111 Karan

222 Aryan

4.5 Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

 Example of Constructor Overloading

```
//Java program to overload constructors
class Student5{
   int id;
   String name;
   int age;
   //creating two arg constructor
   Student5(int i,String n){
   id = i;
   name = n;
   }
   //creating three arg constructor
   Student5(int i,String n,int a){
   id = i;
   name = n;
   age=a;
   }
   void display(){System.out.println(id+" "+name+" "+age);}

   public static void main(String args[]){
   Student5 s1 = new Student5(111,"Karan");
   Student5 s2 = new Student5(222,"Aryan",25);
   s1.display();
   s2.display();
  }
}
```

Output:

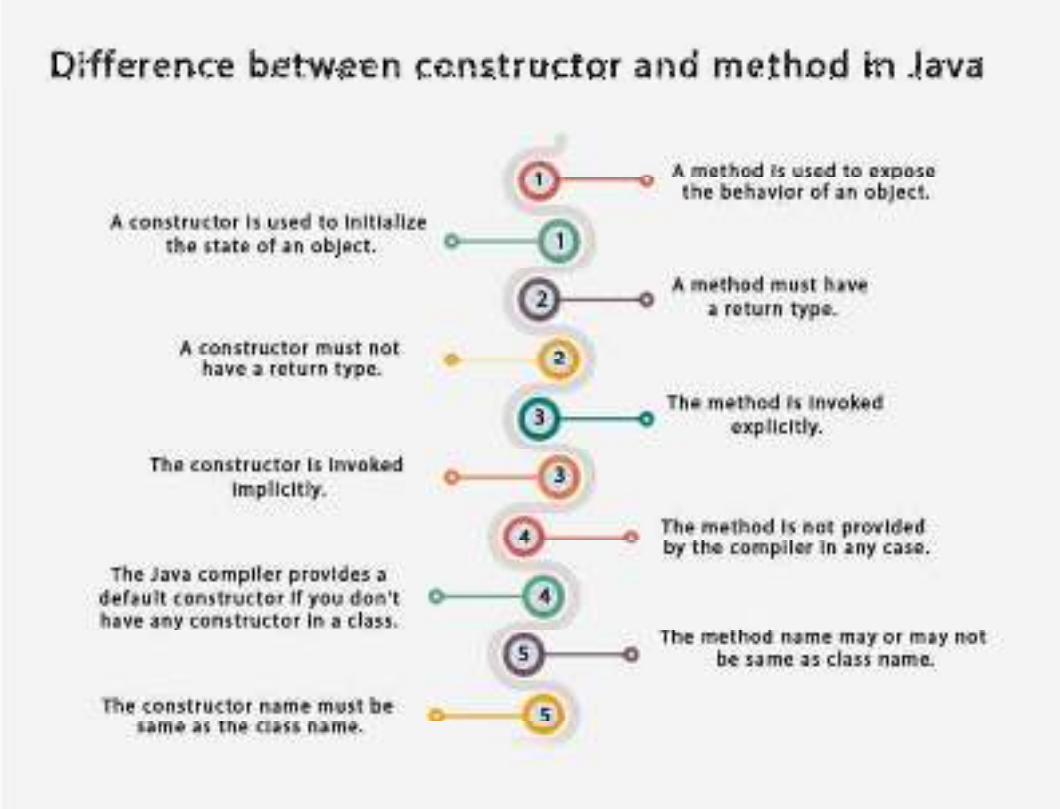
111 Karan 0

222 Aryan 25

4.6 Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.



Java Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in Java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using Java constructor.

```
//Java program to initialize the values from one object to another object.
class Student6{
   int id;
   String name;
   //constructor to initialize integer and string
   Student6(int i,String n){
   id = i;
   name = n;
   }
   //constructor to initialize another object
   Student6(Student6 s){
   id = s.id;
   name =s.name;
   }
   void display(){System.out.println(id+" "+name);}

   public static void main(String args[]){
   Student6 s1 = new Student6(111,"Karan");
   Student6 s2 = new Student6(s1);
   s1.display();
   s2.display();
  }
}
```

Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
class Student7{
   int id;
   String name;
   Student7(int i,String n){
   id = i;
   name = n;
```

```
  }
  Student7(){}
  void display(){System.out.println(id+" "+name);}

  public static void main(String args[]){
  Student7 s1 = new Student7(111,"Karan");
  Student7 s2 = new Student7();
  s2.id=s1.id;
  s2.name=s1.name;
  s1.display();
  s2.display();
 }
}
```

Summary

- In Java, constructors play a crucial role in initializing new objects. They are special methods with the same name as their class and no return type, tasked with setting initial values for object attributes. Java supports two types of constructors: default and parameterized.
- The default constructor is automatically provided if no custom constructors are defined, initializing member variables to default values.
- In contrast, parameterized constructors allow for the explicit initialization of objects with specific values, offering greater flexibility and control over how objects are set up at the moment of their creation.
- The example provided illustrates both constructor types within a Vehicle class context. The default constructor sets the vehicle's model and year to predetermined values, whereas the parameterized constructor accepts arguments to assign specific model and year values to each new Vehicle object.
- This mechanism demonstrates Java's ability to handle object initialization in a straightforward yet powerful manner, ensuring objects are properly configured with relevant state information right from their instantiation.

Keywords

Java Constructor

A constructor in Java is a special block that is called when an object is created. It sets up the object's initial state.

Default Constructor

A default constructor doesn't have any parameters and is automatically provided by Java if no other constructor is defined. It sets an object's attributes to default values.

Parameterized Constructor

A parameterized constructor takes parameters and allows setting specific initial values to an object's attributes when it is created.

Constructor Overloading

Constructor overloading is having more than one constructor in a class, each with a different set of parameters. It allows different ways of initializing an object.

Copy Constructor

A copy constructor is a concept (not built-in in Java) where a new object is created by copying values from an existing object.

Differences Between Constructors and Methods

Purpose: Constructors initialize objects, while methods define what an object can do.

Naming: Constructors must have the same name as the class. Methods can have any name.

Return Type: Constructors do not have a return type. Methods have a return type, which can be void.

Self Assessment

1. What is the size of the int data type in Java?

A. 8 bits
B. 16 bits
C. 32 bits
D. 64 bits

2. Which of the following is used to interpret and execute Java applet Classes hosted by HTML?

A. JVM
B. JRE
C. JDK
D. Web browser

3. Constructor overloading in Java refers to:

A. A class having multiple constructors with different names.
B. A class having multiple constructors with a different number of parameters.
C. A class having multiple methods with the same name.
D. A class having two constructors with the same number of parameters but different types.

4. Which of the following is not a Java feature?

A. Object-oriented
B. Use of pointers
C. Platform-independent
D. Dynamic

5. What does the static keyword in Java indicate?

A. The method belongs to the main class.
B. Variables and methods are independent of any instance of a class.
C. The method can't be modified.
D. The variable values will be stored in static memory.

6. In Java, which method is called when an object is created?

A. init()
B. start()
C. Constructor
D. main()

7. Which access modifier makes a member accessible only within its own class?

A. public
B. private
C. protected
D. default

8. What is the default value of the boolean variable in Java?

A. true
B. false
C. null
D. 0

9. What is inheritance in Java?

A. A method of creating new classes from existing ones.
B. The process of method overloading.
C. The capability of a class to use the properties and methods of another class.
D. Both A and C

10. Which of the following is true about a final class in Java?

A. It can be inherited.
B. It cannot be inherited.
C. It can inherit other classes.
D. It cannot have methods.

11. What is encapsulation in Java?

A. The process of binding data and code together as a single unit.
B. The technique of making the fields in a class private and providing access via public methods.
C. A feature that helps to manage the state of an object.
D. Both A and B

12. What does the extends keyword signify in Java?

A. The class is extending its functionality.
B. The class is inheriting from an interface.
C. The class is inheriting from another class.
D. The method is being overridden.

13. Which interface does java.util.Hashtable implement?

A. List
B. Map
C. Set
D. Hashtable doesn't implement any interfaces

14. Which of these is a mechanism for naming and visibility control of a class and its content?

A. Object
B. Packages
C. Interfaces
D. None of the Above

15. What is the output of the expression 1 + 2 + "3"?

A. 6
B. 33
C. 123
D. Compilation error

Answers for Self Assessment

1.	C	2.	D	3.	B	4.	B	5.	B
6.	C	7.	B	8.	B	9.	D	10.	B
11.	D	12.	C	13.	B	14.	B	15.	B

Review Questions

1. What is a constructor in Java, and how does it differ from a method?
2. Explain the purpose of a default constructor in Java classes.
3. Can a constructor in Java be overridden like methods? Justify your answer.
4. Discuss the role and benefits of parameterized constructors in Java.
5. Explain the concept of constructor overloading in Java with an example.
6. How does the **this** keyword work within constructors? Provide a scenario where it's necessary.

Further Readings

Deitel, P. J., & Deitel, H. M. (2009). Java for programmers. Pearson education.

Kendal, S. (2009). Object oriented programming using Java. Bookboon.

Dale, N. B., Joyce, D. T., & Weems, C. (2002). Object-oriented data structures using Java. Jones & Bartlett Learning.

Dale, N. B., Joyce, D. T., & Weems, C. (2002). Object-oriented data structures using Java.

Jones & Bartlett Learning.

Friesen, J. (2012). Beginning Java 7. Apress.

Fain, Y. (2011). Java programming 24-hour trainer. John Wiley & Sons

Web Links

https://staff.um.edu.mt/__data/assets/pdf_file/0010/57169/jn.pdf

https://staff.um.edu.mt/__data/assets/pdf_file/0010/57169/jn.pdf

Unit 05: String Manipulations

CONTENTS

Objectives

After this unit you will be able to:

- Learn the basic concept of Strings.
- Understand the ways of creating Strings.
- Use different String Functions.
- Implement the various methods of StringBuffer and StringBuilder Class.
- Differentiate between StringBuffer and StringBuilder Class.

Introduction

String manipulation in Java involves performing various operations to modify, extract, or manipulate the content of strings. The String class in Java provides several methods that allow you to manipulate strings effectively. We can define a string as a collection of characters. Java handles character strings by using two final classes, namely, String class and StringBuffer class. The String class is used to implement character strings that are immutable and read-only after the creation and initialization of the string. The StringBuffer class is used to implement dynamic character strings.

5.1 Strings

A string is a sequence of characters but it's not a primitive type. When we create a string in java, it creates an object of type String. A string is an immutable object which means that it cannot be changed once it is created. A string is the only class where operator overloading is supported in java. We can concat two strings using the + operator. Java provides two useful classes for String manipulation – StringBuffer and StringBuilder.

 Example "a"+"b"="ab".

Strings in Java are Objects that are backed internally by a char array. Since arrays are immutable(cannot grow), Strings are immutable as well. Whenever a change to a String is made, an

entirely new String is created. A String variable contains a collection of characters surrounded by double-quotes.

Syntax:

<String_Type> <string_variable> = "<sequence_of_string>";

Example: String str = "Welcome";

Memory Allocation to Strings

JVM divides the allocated memory to a Java program into two parts.

- Stack
- heap.

Stack is used for execution purposes and heap is used for storage purposes. In that heap memory, JVM allocates some memory specially meant for string literals. This part of the heap memory is called String Constant Pool. Whenever you create a string object using a string literal, that object is stored in the string constant pool. Whenever you create a string object using a new keyword, such object is stored in the heap memory.

Example

- String s1 = "abc";
- String s2 = "xyz";
- String s3 = "123";
- String s4 = "A";

will be stored in the String Constant Pool.

When you create string objects using new keyword like below, they will be stored in the heap memory.

Example:

- String s5 = new String("abc");
- char[] c = {'J', 'A', 'V', 'A'};
- String s6 = new String(c);
- String s7 = new String(new StringBuffer());

One more interesting thing about String Constant Pool is that pool space is allocated to an object depending upon its content. There will be no two objects in the pool having the same content.

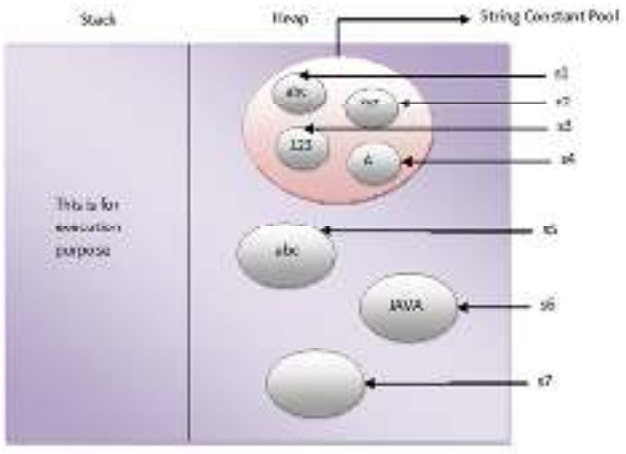


Figure 1: Data in String Constant Pool

Different Ways of Creating Strings

The following are the ways to create a string object:

1. Using string literal.
2. Using a new keyword.

Using string literal

This is the most common way of creating a string. In this case, a string literal is enclosed with double-quotes.

Syntax:

String var_name="value"

Example:

String str = "abc";

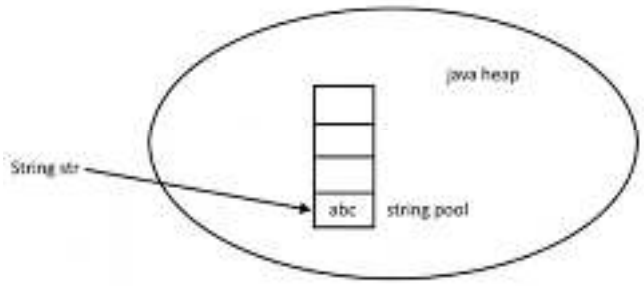


Figure 2:Internal Working

When we create a String using double quotes, JVM looks in the String pool to find if any other String is stored with the same value. If found, it just returns the reference to that String object else it creates a new String object with a given value and stores it in the String pool.

Using new keyword

We can create String object using new operator, just like any normal java class. There are several constructors available in String class to get String from char array, byte array, StringBuffer and StringBuilder.

Example:

String str = new String("abc");

char[ ] a = {'a', 'b', 'c'};

String str2 = new String(a);

Figure 3:Internal Working Using new

Example

```
public class StringDemo {
  public static void main(String args[ ]) {
    char[ ] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
    String helloString = new String(helloArray);
    System.out.println( helloString );
  }
}
```

Output

Hello

5.2 Working with Strings

A String in Java is an object, which contains methods that can perform certain operations on strings which are as follows:

- Location of character
- String Length.
- String Concatenation.
- Finding a Character in a String.
- To convert the case of a String.
- Comparison of Strings.
- contains

Location of Character

The charAt() method in the Java String class is used to retrieve the character at a specified index in a string. The index starts from 0 and goes up to length - 1, where length is the number of characters in the string. If the index is out of bounds (less than 0 or greater than or equal to length), the method throws a StringIndexOutOfBoundsException.

Syntax

public char charAt(int index)

Description: The charAt() method in the Java String class retrieves and returns the character at the specified index position in the string. The index starts from 0 for the first character and goes up to length - 1 for the last character, where length represents the total number of characters in the string. If the index is outside this valid range (i.e., negative or greater than or equal to length), the method throws a StringIndexOutOfBoundsException.

Parameters:

index: The index position of the character to be retrieved. It starts from 0.

Return Value: The character at the specified index position in the string.

Exceptions: StringIndexOutOfBoundsException: Thrown if the index is a negative value or greater than or equal to the length of the string.

 Example:

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

```
String s1="Welcome";
char c=s1.charAt(8);
System.out.println(c);
}
}
```

Output

```
PS D:\392> javac chat.java
PS D:\392> java chat
o
```

String Length

The accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object. Methods used to obtain information about an object are known as accessor methods.

 Example:

```
public class Main {
  public static void main(String[] args) {
    String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    System.out.println("The length of the txt string is: " + txt.length());
  }
}
```

Output: 26

String Concatenation

- Strings are more commonly concatenated with the + operator
- The + operator can be used between strings to combine them.

 Example:

```
"Hello," + " world" + "!“
results in −"Hello, world!"
public class StringDemo {
   public static void main(String args[]) {
      String string1 = "saw I was ";
      System.out.println("Dot " + string1 + "Tod");
   }
}
```

Output:

```
Dot saw I was Tod
```

You can also use the concat() method with string literals for concatenation.

Syntax:

```
string1.concat(string2);
```

This returns a new string that is string1 with string2 added to it at the end.

 Example:

"My name is ".concat("Zara");

Finding a Character in a String

The indexOf() method returns the index (the position) of the first occurrence of a specified text in a string (including whitespace).

 Example:

```
public class Main {
  public static void main(String[] args) {
    String txt = "Please locate where 'locate' occurs!";
    System.out.println(txt.indexOf("locate"));
  }
}
```

Output: 7 // Java counts positions from zero. 0 is the first position in a string, 1 is the second, 2 is the third .

To convert the case of a String

toUpperCase() and toLowerCase() are used to convert the case of a string.

 Example:

```
String txt = "Hello World";
System.out.println(txt.toUpperCase());   // Outputs "HELLO WORLD"
System.out.println(txt.toLowerCase());   // Outputs "hello world"
```

Comparison of Strings

String class provides equals() and equalsIgnoreCase() methods to compare two strings. These methods compare the value of a string to check if two strings are equal or not. It returns true if two strings are equal and false if not.

 Example

```
public class StringEqualExample
{
 public static void main(String[] args)
{ //creating two string object
 String s1 = "abc";
String s2 = "abc";
String s3 = "def";
String s4 = "ABC";
System.out.println(s1.equals(s2));//true
System.out.println(s2.equals(s3));//false
System.out.println(s1.equals(s4));//false;
System.out.println(s1.equalsIgnoreCase(s4));//true } }
```

String class implements Comparable interface, which provides compareTo() and compareToIgnoreCase() methods and it compares two strings lexicographically. Both strings are

converted into Unicode values for comparison and return an integer value that can be greater than, less than, or equal to zero. If strings are equal then it returns zero or else it returns either greater or less than zero.

 Example

```
public class StringCompareToExample {
public static void main(String[] args)
{
String a1 = "abc";
String a2 = "abc";
String a3 = "def";
String a4 = "ABC";
System.out.println(a1.compareTo(a2));//0
System.out.println(a2.compareTo(a3));//less than 0
System.out.println(a1.compareTo(a4));//greater than 0
System.out.println(a1.compareToIgnoreCase(a4));//0
}}
```

Contains

Java String contains() methods that check if the string contains a specified sequence of characters or not. This method returns true if the string contains a specified sequence of characters, else returns false.

 Example

```
public class StringContainsExample
{
 public static void main(String[] args)
 {
String s = "Hello World"; System.out.println(s.contains("W"));//true
System.out.println(s.contains("X"));//false
 }}
```

String Array

As the name suggests, a string array is an array containing strings. We can declare string arrays in the following two ways:

1. With an initial size

2. Without an initial size

1. With an Initial Size:

In Java, we can declare string arrays and assign an initial size to them.

```
public StringArrayDemo
{
private String[ ] habit = new String[10];
// more to the class here ...
}
```

```
void populateStringArray( )
{
habit[0] = “Hello";
habit[1] = “ Welcome";
habit[2] = "Great";
// ...
}
```

In this example,

1. A class StringArrayDemo is created and declared public using the public keyword.
2. In the class JavaStringArrayDemo, a String array named as fruits is created, where the habit array has been given an initial size of 10 elements.
3. Then, the elements in the String array are assigned by the populateStringArray( ) method in the class:

(a) habit[0] is assigned a string Hello.

(b) habit[1] is assigned a string Welcome.

(c) habit[2] is assigned a string Great.

Did you Know? A String array in Java begins with an element numbered zero.

2. Without an Initial Size:

We can also declare a Java String array without giving it an initial size.

```
public class JavaStringArrayDemo
{
private String[ ] toppings;
// more to the class here ...
}
```

After this, the Java array can be given size in the program code, and populated as desired, like this:

```
void populateStringArray( )
{
fruits[0] = "Apple";
fruits[1] = "Mango";
fruits[2] = "Banana";
// ...
}
```

This method of declaring an array is very similar to the first method. However, in this method, the string array is not given any size until the populateStringArray method is called.

 **Did you Know?** A String array in Java begins with an element numbered zero.

5.3 StringBuffer Class

Java StringBuffer class is used to create a mutable string. This class is the same as the String class except it is mutable i.e. it can be changed. StringBuffer may have characters and substrings inserted in the middle or appended to the end. Following are the important points about StringBuffer:

- A string buffer is like a String but can be modified.
- It contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.
- They are safe for use by multiple threads.
- Every string buffer has a capacity.

Table 1: StringBuffer Constructors

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	creates a string buffer with the specified string.
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length.

 Example: StringBuffer s=new StringBuffer();

StringBuffer(int size): It accepts an integer argument that explicitly sets the size of the buffer.

 Example: StringBuffer s=new StringBuffer(20);

StringBuffer(String str): It accepts a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

 Example: StringBuffer s=new StringBuffer("Welcome");

The following table shows the various methods used in the StringBuffer class:

Table 2: StringBuffer Constructors

Method	Description
append(String s)	It is used to append the specified string with this string.
insert(int offset, String s)	It is used to insert the specified string with this string at the specified position.
replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.
reverse()	It is used to reverse the string.
capacity()	It is used to return the current capacity.
ensureCapacity(int minimumCapacity)	It is used to ensure the capacity is at least equal to the given minimum.
charAt(int index)	It is used to return the character to the specified position.

length()	It is used to return the length of the string i.e. total number of characters.
substring(int beginIndex)	It is used to return the substring from the specified beginIndex.
substring(int beginIndex, int endIndex)	It is used to return the substring from the specified beginIndex and endIndex.

 Example

```
//Program to implement StringBuffer append() method
class StringBufferExample{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
}
}
```

 Example

```
// Program to implement StringBuffer insert() method
class StringBufferExample2{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello
}
}
```

 Example

```
// Program to implement StringBuffer replace() method
class StringBufferExample3{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.replace(1,3,"Java");
System.out.println(sb); //prints HJavalo
}
}
```

 Example

```
// Program to implement StringBuffer delete() method
class StringBufferExample4{
```

```
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.delete(1,3);
System.out.println(sb);//prints Hlo
}
}
```

 Example

```
// Program to implement StringBuffer reverse() method
class StringBufferExample5{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.reverse();
System.out.println(sb); //prints olleH
}
}
```

 Example

```
// Program to implement StringBuffer capacity() method
class StringBufferExample6{
public static void main(String args[]){
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity()); //default 16
sb.append("Hello");
System.out.println(sb.capacity()); //now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity()); //now (16*2)+2=34 i.e (oldcapacity*2)+2
}
}
```

 Example

```
// Program to implement StringBuffer ensureCapacity() method
class StringBufferExample7{
public static void main(String args[]){
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity()); //now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity()); //now (16*2)+2=34 i.e (oldcapacity*2)+2
sb.ensureCapacity(10); //now no change
```

```
System.out.println(sb.capacity()); //now 34
sb.ensureCapacity(50);  //now (34*2)+2
System.out.println(sb.capacity()); //now 70
}
}
```

5.4 StringBuilder Class

Java StringBuilder class is used to create a mutable (modifiable) string. The Java StringBuilder class is the same as the StringBuffer class except that it is non-synchronized. It is available since JDK 1.5. StringBuilder class provides an API similar to StringBuffer, but unlike StringBuffer, it doesn't guarantee thread safety. The following table shows the various constructors of the StringBuilder class:

Table 3: StringBuilder Constructors

Constructor	Description
StringBuilder()	Creates an empty string builder with a default capacity of 16 (16 empty elements).
StringBuilder(CharSequence cs)	Constructs a string builder containing the same characters as the specified CharSequence, plus an extra 16 empty elements trailing the CharSequence.
StringBuilder(int initCapacity)	Creates an empty string builder with the specified initial capacity.
StringBuilder(String s)	Creates a string builder whose value is initialized by the specified string, plus an extra 16 empty elements trailing the string.

StringBuilder Methods

The following are the various methods used by StringBuilder class:

StringBuilder Length and Capacity

```
// creates empty builder, capacity 16
StringBuilder sb = new StringBuilder();
// adds 5 character string at beginning sb.append("Hello");
System.out.println("StringBuilder length = "+sb.length()); // prints 5
System.out.println("StringBuilder capacity = "+sb.capacity()); // prints 16
```

Append()

```
public class StringBuilderExample
{
public static void main(String[] args)
{
StringBuilder sb = new StringBuilder("Hello "); sb.append("World");// now original string is changed
System.out.println(sb);// prints Hello World
}
```

}

Insert()

```
StringBuilder sb = new StringBuilder("HellWorld");
sb.insert(4, "o ");
System.out.println(sb); // prints Hello World
```

replace(int startIndex, int endIndex, String str)

```
StringBuilder sb = new StringBuilder("Hello World!");
sb.replace(6,11,"Earth");
System.out.println(sb); // prints Hello Earth!
```

delete(int startIndex, int endIndex)

```
StringBuilder sb = new StringBuilder("Journalgood.com");
sb.delete(7,14);
System.out.println(sb); // prints Journal
```

Capacity()

```
StringBuilder sb=new StringBuilder();
System.out.println(sb.capacity()); // default value 16
sb.append("Java");
System.out.println(sb.capacity()); // still 16
sb.append("Hello StringBuilder Class!");
System.out.println(sb.capacity()); // (16*2)+2
```

Reverse()

```
StringBuilder sb = new StringBuilder("lived");
sb.reverse();
System.out.println(sb);// prints devil
```

Table 4: StringBuffer v/s StringBuilder

StringBuffer	StringBuilder
StringBuffer is *synchronized* i.e. thread-safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-synchronized* i.e., not thread-safe. It means two threads can call the methods of StringBuilder simultaneously.
Operates slower due to thread safety feature	Better performance compared to StringBuffer
Has some extra methods – substring, length, capacity, etc.	Not needed because these methods are present in String too.
Introduced in Java 1.2	Introduced in Java 1.5 for better performance.
StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.

Summary

- A string is a sequence of characters that is created using the String class.
- The length of a string refers to the number of characters in a string.

- An array that contains strings is a string array.
- The string class consists of some methods for creating string objects. These methods are called String methods.
- The StringBuffer class is a peer class of String, which is used for strings' alteration.

Keywords

String: It is a keyword used to declare and manipulate strings.

new: The new keyword is used to create a new instance of an object, including strings.

StringBuilder(): Creates an empty string builder with a default capacity of 16 (16 empty elements).

concat(): The String concat() method combines a specific string at the end of another string and ultimately returns a combined string.

StringBuilder(): Creates an empty string builder with a default capacity of 16 (16 empty elements).

append: The append method is used to add characters or other data to the end of the sequence.

insert: The insert method is used to insert characters or other data at a specified index in the sequence.

Self Assessment

1. Which of the following methods is used to create object a's string representation?

A. a.toString( )
B. StringTokenizer.countTokens( )
C. str1.append (str2)
D. str2 = str1.replace('a', 'b');

2. Which of these methods of String class can be used to test strings for equality?

A. isequal()
B. isequals()
C. equal()
D. equals()

3. What will be the output of the following Java program?

```
class string_demo
{
   public static void main(String args[])
   {
      String obj = "I" + "like" + "Java";
      System.out.println(obj);
   }
}
```

A. I
B. like
C. Java

D. IlikeJava

4. What will be the output of the following Java program?

```
class string_class
{
   public static void main(String args[])
   {
      String obj = "I LIKE JAVA";
      System.out.println(obj.charAt(3));
   }
}
```

A. I
B. L
C. K
D. E

5. Which of these operators can be used to concatenate two or more String objects?

A. +
B. +=
C. &
D. ||

6. Which of this methods of class String is used to obtain a length of String object?

A. get()
B. Sizeof()
C. lengthof()
D. length()

7. Which of these classes is a superclass of String and StringBuffer class?

A. java.util
B. java.lang
C. ArrayList
D. None of the mentioned

8. What is the value returned by unction compareTo() if the invoking string is less than the string compared?

A. zero
B. a value less than zero
C. a value greater than zero
D. None of the mentioned

9. Which of these data type value is returned by the equals() method of String class?

A. char
B. int
C. boolean
D. All of the mentioned

10. What is the correct way to compare two strings for equality in Java?

A. str1 == str2
B. str1.equals(str2)
C. str1.compare(str2) == 0
D. str1.compareTo(str2)

11. Which method is used to concatenate two strings in Java?

A. concat()
B. combine()
C. merge()
D. append()

12. What is the purpose of the length() method in the String class?

A. Returns the number of characters in a string.
B. Returns the index of a specified character.
C. Returns the substring of a string.
D. Returns the uppercase version of a string.

13. Which method is used to convert a string to lowercase in Java?

A. toLowerCase()
B. convertToLower()
C. toLower()
D. changeCaseToLower()

14. What does the indexOf() method in the String class return if the specified substring is not found?

A. -1
B. 0
C. Throws an exception
D. Returns 1

15. Which one of the following is a valid statement?

A. char[] c = new char();
B. char[] c = new char[5];
C. char[] c = new char(4);
D. char[] c = new char[];

Answers for Self Assessment

1.	D	2.	D	3.	D	4.	A	5.	A
6.	D	7.	B	8.	B	9.	C	10.	D
11.	A	12.	A	13.	A	14.	A	15.	B

Review Questions

1. Explain the process of concatenating two strings in Java. Provide examples and discuss any alternative methods for string concatenation.
2. Choose three common methods from the String class (e.g., length(), substring(), indexOf()) and explain their usage. Provide examples to demonstrate how these methods work and when they might be useful.
3. Compare and contrast the StringBuilder class with the String class in terms of mutability and performance. In what situations would you prefer to use one over the other?
4. "Various string methods are used for different tasks of string manipulation." Discuss those methods with examples.
5. "The StringBuffer class consists of some methods that can be used for the manipulation of the objects of that class." Elaborate.

Further Readings

Balagurusamy E. Programming with Java 3e Primer. New Delhi: Tata McGraw Publishers.

Schildt H. Java A Beginner's Guide, 3rd ed. New York: Mc-Graw Hill.

Arnold, K., Gosling, J., & Holmes, D. (2005). *The Java programming language*. Addison Wesley Professional.

Haggar, P. (2000). *Practical Java: programming language guide*. Addison-Wesley Professional. Online

Web Links

http://www.javabeginner.com/learn-java/java-string-comparison

http://www.leepoint.net/notesjava/data/strings/55stringTokenizer/10stringtokenizer.html

http://admashmc.com/main/images/Lec_Notes/javaarray.pdf

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

https://www.w3schools.com/java/java_arrays.asp

https://www.baeldung.com/java-arrays-guide

Unit 06: Inheritance & Interfaces

CONTENTS

Objectives

After this unit students will be able to:

- Learn the concept and different types of inheritance.
- Understand the concept of super class and sub-class.
- Implementation of method overriding.

Introduction

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows classes to inherit attributes and methods from other classes. It enables code reusability and promotes a hierarchical relationship among classes. At its core, inheritance embodies the idea of a "parent-child" relationship, where a subclass (child class) can inherit properties and behaviors from a superclass (parent class). This means that the subclass automatically has access to all the fields and methods of its superclass.

6.1 Overview Of Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) where a new class, referred to as the subclass or derived class, can inherit attributes and behaviors (methods) from an existing class, known as the superclass or base class. This enables code reusability and promotes the creation of hierarchical relationships among classes.

The following are some of the key terminologies of inheritance:

Superclass/Base Class/Parent Class: This is the existing class from which other classes derive attributes and behaviors. It serves as a template or blueprint for subclasses. The superclass encapsulates common properties and behaviors shared by its subclasses.

Subclass/Derived Class/Child Class: This is the new class that inherits attributes and behaviors from the superclass. It can add its own unique attributes and behaviors, as well as override or extend the functionality of the superclass methods.

Syntax: In most programming languages that support OOP, inheritance is implemented using syntax like extends (Java, JavaScript, PHP), : (Python, Swift), or < (C++).

Example

Java: class Subclass extends Superclass { ... }

Python: class Subclass(Superclass): .

6.2 Types of Inheritance

Single Inheritance: A subclass inherits from only one superclass.

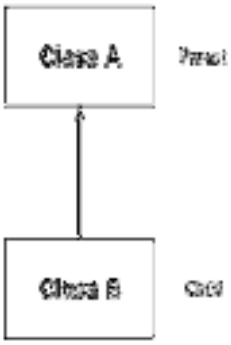


Figure 1: Single Level Inheritance

Multiple Inheritance: A subclass inherits from more than one superclass. Some languages like C++ support multiple inheritance, while others like Java do not directly support it.

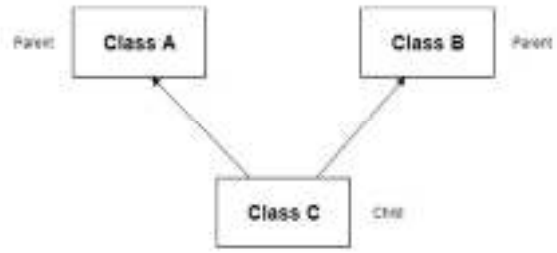


Figure 2:Multiple Inheritance

Note: In Java Multiple inheritance is not supported because it may become ambiguous in case if more than one parent class have same type of method.

Multilevel Inheritance: Subclasses can derive from other subclasses, forming a hierarchy.

Figure 3Multilevel Inheritance

Hierarchical Inheritance: Multiple subclasses are inherited from a single superclass.

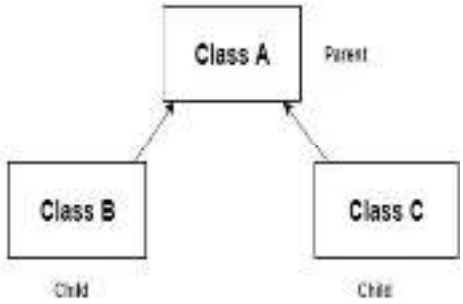


Figure 4: Hierarchical Inheritance

Hybrid Inheritance: A combination of multiple types of inheritance.

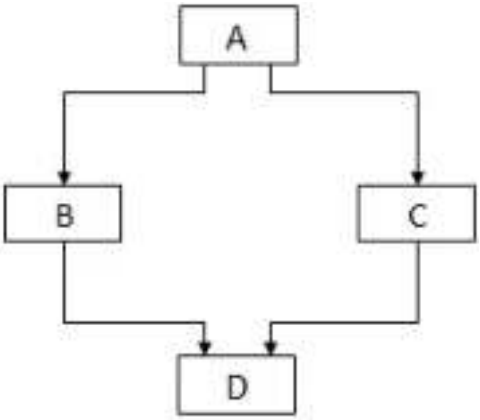


Figure 5:Hybrid Inheritance

Access Modifiers: Inheritance can also involve access control, where subclasses may have different levels of access to superclass members depending on their visibility. Common access modifiers include public, protected, and private.

Method Overriding: Subclasses can provide a specific implementation of a method that is already defined in the superclass. This is called method overriding. It allows subclasses to tailor the behavior of inherited methods to suit their specific requirements.

Method Overloading: Some languages allow method overloading, where a subclass can define multiple methods with the same name but different parameter lists. Overloaded methods within a class have the same name but different signatures.

Polymorphism: Inheritance facilitates polymorphism, which allows objects of different classes to be treated as objects of a common superclass. This enables more flexible and dynamic programming.

Inheritance is a powerful mechanism in OOP that promotes code reuse, modularization, and abstraction, making it easier to manage and maintain complex software systems. However, misuse of inheritance can lead to tight coupling between classes and can make the codebase harder to understand and maintain.

Example:

```
import java.util.*;

class base

{

Scanner s=new Scanner(System.in);
```

```
int roll=s.nextInt();
}
class derived extends base
{
Scanner s1=new Scanner(System.in);
String name=s1.nextLine();
}
class mul extends derived
{
Scanner s2=new Scanner(System.in);
int age=s2.nextInt();
}
class imp
{
public static void main(String args[])
{
derived o=new derived();
System.out.println("Roll= "+o.roll);
System.out.println("Name= "+o.name);
System.out.println("Age= "+o.age);
}
}
PS D:\392\Programs> javac imp.java
PS D:\392\Programs> java imp
12
hello
23
Roll= 12
Name= hello
Age= 23
```

Note: A subclass inherits all the members, including fields, methods, and nested classes, from its superclass. Constructors, however, are not considered members, and therefore they are not inherited by subclasses. Nevertheless, the constructor of the superclass can be invoked from the subclass.

6.3 Working With Subclasses and Super Classes

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

subclass (child) - the class that inherits from another class

superclass (parent) - the class being inherited from

To inherit from a class, use the extends keyword.

In Java, working with subclasses and superclass's involves creating classes that extend other classes to inherit their attributes and methods. Here's an example demonstrating the basics of subclassing and superclassing in Java:

 Example

```
// Superclass
class Animal {
    private String name;

    public Animal(String name) {
        this.name = name;
    }
    public void eat() {
        System.out.println(name + " is eating.");
    }

    public void sleep() {
        System.out.println(name + " is sleeping.");
    }
}
// Subclass
class Dog extends Animal {
    public Dog(String name) {
        super(name); // Call superclass constructor
    }

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

// Main class
public class Main {
    public static void main(String[] args) {
        // Create an instance of the superclass
        Animal animal = new Animal("Generic Animal");
        animal.eat();
        animal.sleep();

        // Create an instance of the subclass
        Dog dog = new Dog("Buddy");
```

```
    dog.eat(); // Inherited from superclass
    dog.sleep(); // Inherited from superclass
    dog.bark(); // Specific to Dog subclass
  }
}
```

Explanation:

Animal is the superclass, containing common behaviors and attributes shared among different animals.

Dog is a subclass of Animal, inheriting behaviors and attributes from it. Additionally, Dog has its own unique behavior bark().

In the Main class, instances of both superclass (Animal) and subclass (Dog) are created and their methods are invoked.

Output:

Generic Animal is eating.

Generic Animal is sleeping.

Buddy is eating.

Buddy is sleeping.

Woof! Woof!

This demonstrates how subclasses inherit methods from superclasses and can also have their own unique methods. Additionally, constructors in the superclass can be called using super() within the subclass constructor.

Note: Polymorphism is a feature in OOPs, which uses inherited methods to perform different tasks.

Use of final keyword

The final keyword to prevent a class from being subclassed. When a class is declared as final, it cannot be extended by any other class. Here's an example demonstrating the use of a final class in inheritance:

Example

```
// Final superclass
final class Animal {
  public void makeSound() {
    System.out.println("Animal makes a sound");
  }
}

// This will cause a compilation error since Dog cannot inherit from a final class
// class Dog extends Animal {
//    public void makeSound() {
//        System.out.println("Dog barks");
//    }
// }
```

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


      // Dog class cannot be defined because Animal is final
   }
}
```

Explanation:

The Animal class is declared as final, indicating that it cannot be subclassed.

Attempting to create a subclass of Animal, such as Dog, will result in a compilation error because Animal is marked as final.

Output:

Animal makes a sound

This demonstrates how you can use a final class to prevent inheritance in Java. By marking a class as final, you ensure that its behavior and implementation cannot be altered or extended by other classes.

6.4 Overriding Methods in The Super Class

In Java, overriding is a mechanism that enables a subclass or child class to supply its own implementation of a method that is already defined by one of its superclasses or parent classes. This means that when a method in a subclass shares the same name, parameter list (or signature), and return type (or a subtype) as a method in its superclass, it effectively replaces or overrides the behavior of the superclass method.

In Java, you can override methods defined in a superclass within its subclass. This allows you to provide a specific implementation of a method in the subclass that is different from the implementation in the superclass. In Java, method overriding is a key mechanism for achieving runtime polymorphism. When a method is overridden in a subclass, the version of the method executed is determined by the type of object used to invoke it. If the method is invoked using an object of the parent class, the version in the parent class will be executed. Conversely, if the method is invoked using an object of the subclass, the version in the subclass will be executed. In essence, it's the actual type of the object being referred to—not the type of the reference variable—that dictates which overridden method will be executed.

Here's how you can override methods in Java:

 Example

```
// Superclass
class Animal {
   public void makeSound() {
      System.out.println("Animal makes a sound");
   }
}
// Subclass
class Dog extends Animal {
```

```
    @Override // Annotation indicating method overriding (optional but recommended)
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

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

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

Explanation:

The Animal class defines a method makeSound().

The Dog class extends Animal and overrides the makeSound() method with its own implementation.

In the Main class, instances of both Animal and Dog are created. When calling the makeSound() method on each instance, the appropriate implementation is executed based on the object's type.

Output:

Animal makes a sound

Dog barks

This demonstrates method overriding in Java, where the subclass provides its own implementation of a method defined in the superclass. By using method overriding, you can tailor the behavior of inherited methods to suit the specific requirements of subclasses.

Rules for Java Method Overriding

1. Overriding and Access Modifiers

The access modifier for an overriding method can allow more, but not less, access than the overridden method.

 Example, a protected instance method in the superclass can be made public, but not private, in the subclass. Doing so will generate a compile-time error.

2. Final methods can not be overridden

If we don't want a method to be overridden, we declare it as final. Please see Using Final with Inheritance.

 Example

```
// A Java program to demonstrate that
```

```
// final methods cannot be overridden


class Parent {
        // Can't be overridden
        final void show() {}
}


class Child extends Parent {
        // This would produce error
        void show() {}
}
```

Output

```
error: show() in Child cannot override show() in Parent
   void show() {  }
      ^
  overridden method is final
```

3. Static methods can not be overridden(Method Overriding vs Method Hiding):

When you define a static method with the same signature as a static method in the base class, it is known as method hiding. The following table summarizes what happens when you define a method with the same signature as a method in a super-class.

4. Private methods can not be overridden

Private methods cannot be overridden as they are bonded during compile time. Therefore we can't even override private methods in a subclass.

5. The overriding method must have the same return type (or subtype)

From Java 5.0 onwards it is possible to have different return types for an overriding method in the child class, but the child's return type should be a sub-type of the parent's return type. This phenomenon is known as the covariant return type.

6. Invoking overridden method from sub-class

We can call the parent class method in the overriding method using the super keyword.

 Example

```
// Base Class
class Parent {
   void show() { System.out.println("Parent's show()"); }
}


// Inherited class
class Child extends Parent {
   // This method overrides show() of Parent
   @Override void show()
   {
      super.show();
```

```
        System.out.println("Child's show()");
    }
}

// Driver class
class Main {
    public static void main(String[] args)
    {
        Parent obj = new Child();
        obj.show();
    }
}
```

Output

```
Parent's show()
Child's show()
```

Summary

- Java supports single inheritance, meaning a class can inherit from only one superclass
- Java has access modifiers like public, private, protected, and default. Subclasses can access public and protected members of the superclass, but not private members.
- Subclasses can override methods from the superclass to customize their behavior and is achieved by defining a method with the same signature in the subclass.
- The "super" keyword is used to refer to the superclass. It can be used to access superclass members, call superclass constructors, and invoke overridden methods from the superclass.
- Java supports a hierarchical class structure where classes can inherit attributes and methods from their superclass(es).
- Method overriding is a feature in Java that allows a subclass to provide a specific implementation of a method that is already defined in its superclass.
- Method overriding is only applicable to methods in a subclass that inherit from a superclass.
- When overriding a method, the access modifier in the subclass method can be the same as or less restrictive than the access modifier in the superclass method.
- Inside the overriding method, the super keyword can be used to call the superclass version of the method, allowing for additional behavior to be added to the overridden method.

Keywords

extends: Used to establish an inheritance relationship between classes.

super: Keyword used to refer to the superclass from within a subclass. It can be used to access superclass methods, constructor, and fields.

this: Keyword used to refer to the current instance of the class. It can be used to access class fields, methods, or constructors.

@Override: Annotation used to indicate that a method in a subclass is intended to override a method in the superclass.

final: Modifier used to prevent a class from being subclassed or a method from being overridden.

abstract: Modifier used to declare an abstract class or method. Abstract classes cannot be instantiated directly and may contain abstract methods that must be implemented by subclasses.

protected: Access modifier used to restrict access to members (fields, methods, constructors) to within the same package or subclasses. Protected members can be accessed by subclasses even if they are in different packages.

public: Access modifier used to declare members (fields, methods, constructors) that are accessible from any other class.

private: Access modifier used to restrict access to members only within the same class. Private members cannot be accessed by subclasses.

Single Inheritance: This is the simplest form of inheritance where a subclass inherits from only one superclass.

Hybrid Inheritance: It can include any combination of single, hierarchical, multilevel, and multiple inheritance.

Self Assessment

1. What is inheritance in Java?

A. A way to create new classes based on existing classes.

B. A way to create new methods in a class.

C. A way to define interfaces in Java.

D. A way to create objects from classes.

2. Which keyword is used to establish an inheritance relationship between classes in Java?

A. class

B. extends

C. implements

D. inherit

3. Which type of inheritance allows a class to inherit from multiple interfaces?

A. Single inheritance

B. Hierarchical inheritance

C. Multilevel inheritance

D. Multiple inheritance

4. In Java, what is the purpose of the @Override annotation?

A. To indicate that a method is abstract.

B. To indicate that a method is final and cannot be overridden.

C. To indicate that a method is intended to override a method in the superclass.

D. To indicate that a method is private and cannot be accessed outside the class.

5. Which type of inheritance involves a chain of inheritance where a subclass inherits from another subclass?

A. Single inheritance

B. Hierarchical inheritance

C. Multilevel inheritance

D. Hybrid inheritance

6. What is method overriding in Java?

A. Creating a new method in a subclass with the same name as a method in the superclass.
B. Creating a new method in a subclass with a different name from a method in the superclass.
C. Creating a new method in a superclass with a different name from a method in the subclass.
D. Creating a new method in a superclass with the same name as a method in the subclass.

7. Which keyword is used to indicate that a method is intended to override a method in the superclass?

A. override
B. extends
C. implements
D. @Override

8. In method overriding, the subclass method must have the same _____ as the method in the superclass.

A. Name and return type
B. Name and parameters
C. Return type and parameters
D. Name, return type, and parameters

9. When a method is overridden in a subclass, which version of the method is executed when called on an object of the subclass?

A. The version of the method in the superclass
B. The version of the method in the subclass
C. Both versions of the method
D. None of the above

10. Which access modifier can be used for an overriding method to increase its visibility compared to the overridden method in the superclass?

A. private
B. protected
C. default
D. public

11. Which type of inheritance allows a class to inherit from only one superclass?

A. Multiple Inheritance
B. Single Inheritance
C. Hierarchical Inheritance
D. Multilevel Inheritance

12. In Java, what keyword is used to establish an inheritance relationship between classes?

A. inherit
B. implements
C. extends
D. derive

13. Which type of inheritance involves a class inheriting from multiple interfaces?
A. Single Inheritance
B. Hierarchical Inheritance
C. Multilevel Inheritance
D. Multiple Inheritance

14. In multilevel inheritance, a subclass inherits from:
A. Two or more superclasses
B. One superclass and one or more interfaces
C. Another subclass
D. Both a superclass and an interface

15. What is the term used for a combination of different types of inheritance, such as single, hierarchical, and multiple?
A. Complex Inheritance
B. Compound Inheritance
C. Hybrid Inheritance
D. Extended Inheritance

Answers for Self Assessment

1.	A	2.	B	3.	D	4.	C	5.	C
6.	A	7.	D	8.	D	9.	B	10.	D
11.	B	12.	C	13.	D	14.	C	15.	C

Review Questions

1. Discuss the concept of inheritance in Java, its significance in object-oriented programming, and its practical implications.
2. Explain the syntax used for inheritance, and types of inheritance supported in Java, and provide examples to illustrate each type.
3. Discuss the concept of dynamic method dispatch in the context of method overriding.
4. Describe how access modifiers affect method overriding in Java.
5. Explain the requirements for method overriding in Java with the help of an appropriate example.
6. How does method overriding differ from method overloading?

Further Readings

Li, L. (2012). Java: data structures and programming. Springer Science & Business Media.

Eckel, B. (2003). Thinking in JAVA. Prentice Hall Professional.

Liang, Y. D. (2003). Introduction to Java programming. Pearson Education India.

Reges, S., & Stepp, M. (2014). Building Java Programs. Pearson.

Poo, D., Kiong, D., & Ashok, S. (2007). Object-oriented programming and Java. Springer Science & Business Media.

Web Links

https://www.programiz.com/java-programming/inheritance

https://www.tutorialspoint.com/java/java_inheritance.htm

https://docs.oracle.com/javase%2Ftutorial%2F/java/IandI/subclasses.html

https://www.digitalocean.com/community/tutorials/inheritance-java-example

https://www.mygreatlearning.com/blog/inheritance-in-java/

https://runestone.academy/ns/books/published/csawesome/Unit9-Inheritance/topic-9-3-overriding.html

https://ioflood.com/blog/method-overriding-in-java/

Unit 07: More on Inheritance

CONTENTS

Objective

After this unit you will be able to:

- Learn the concept of abstract classes and their implementation.
- Understand how interfaces are used with abstract classes.
- Implement multiple inheritance using interfaces.

Introduction

Inheritance is a fundamental concept in object-oriented programming that allows classes to inherit properties and behaviors from other classes. In Java, inheritance enables code reuse, promotes modularity, and facilitates the creation of class hierarchies. While the basics of inheritance, such as subclassing and method overriding, are essential to understand, there are additional aspects and advanced topics that warrant exploration for a deeper understanding of inheritance in Java. This introduction serves as a gateway to delve into more advanced topics related to inheritance in Java. In the following sections, we will explore various aspects, including:

Inheritance Hierarchies: Understanding how classes are organized into hierarchical structures, including superclasses, subclasses, and their relationships. We will explore how inheritance hierarchies provide a way to model real-world entities and concepts in a systematic and organized manner.

Method Overriding and Polymorphism: Delving deeper into method overriding, dynamic method dispatch, and polymorphism—the ability of objects of different classes to be treated as objects of a common superclass. We will explore how polymorphism enables flexibility, extensibility, and modularity in Java programs.

Abstract Classes and Interfaces: Understanding abstract classes and interfaces as mechanisms for defining contracts, providing abstractions, and promoting code reuse. We will discuss when to use abstract classes versus interfaces and explore how they complement each other in Java inheritance.

Final Classes and Methods: Examining the concept of final classes and methods and their implications for inheritance. We will discuss how finality restricts subclassing and method overriding and explore scenarios where final classes and methods are appropriate.

7.1 Creating And Extending Abstract Classes

In Java, an abstract class is a class that cannot be instantiated on its own but can contain abstract methods, which are methods without a body. Abstract classes are designed to be extended by subclasses, which provide implementations for the abstract methods. Here are some key points about abstract classes in Java:

Syntax:

To declare an abstract class, you use the abstract keyword in the class declaration. Abstract classes may or may not contain abstract methods.

abstractclassMyAbstractClass{ // Abstract methodabstractvoidmyAbstractMethod(); // Concrete methodvoidmyConcreteMethod(){ // Method body} }

Abstract Methods: Abstract methods are declared without a body and end with a semicolon instead of braces. Subclasses of an abstract class must provide implementations for all abstract methods unless they themselves are declared as abstract.

Instantiation: Abstract classes cannot be instantiated directly with the new keyword. You can only create instances of concrete subclasses of the abstract class.

Abstract class in Java

A class which is declared as abstract is known as an abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Rules for Abstract Classes

- An instance of an abstract class cannot be created directly.
- Constructors are allowed in abstract classes. They can be used to initialize fields or perform other necessary setup tasks.
- It's possible to have an abstract class without any abstract methods. Abstract classes can also contain concrete methods and fields.
- A final method can be present in an abstract class, but an abstract method cannot be declared as final. Mixing abstract and final modifiers in the same method declaration is not allowed.
- Static methods can be defined in an abstract class. They are not inherited by subclasses but can be called using the class name.
- The abstract keyword can be used to declare both top-level classes (outer classes) as well as inner classes as abstract.
- If a class contains at least one abstract method, it should be declared as abstract itself. This ensures that the class cannot be instantiated directly.
- If a subclass cannot provide implementations for all abstract methods inherited from its parent abstract class, then it should also be declared as abstract. This allows subsequent subclasses to provide implementations for the remaining abstract methods. This ensures that every concrete subclass ultimately provides implementations for all inherited abstract methods.

 Example

```
// concept of abstract class
abstract class info
{
abstract void show();
}
```

```
class ex extends info
{
void show()
{
System.out.println("Implementation of abstract class");
}
public static void main(String args[])
{
info o=new ex();
o.show();
}
}
```

Output

```
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> javac ex.java
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> java ex
Implementation of abstract class
```

 Example

```
abstract class detail
{
abstract void show();
}
class stu extends detail
{
void show()
{
String name="Seerat";
int age=12;
System.out.println("Entered name is  "+name);
System.out.println("Entered age is "+age);
}
}
class abc
{
public static void main(String args[])
{
detail d=new stu();
d.show();
}
}
```

Output

```
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> javac abc.java
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> java abc
Entered name is  Seerat
Entered age is 1
```

 Example

```
// Extending abstract class
abstract class shape
{
abstract double area();
}
class cir extends shape
{
double r;
cir(double r)
{
this.r=r;
}
double area()
{
return 3.14*r*r;
}
}
class rec extends shape
{
double l;
double b;
rec(double l, double b)
{
this.l=l;
this.b=b;
}
double area()
{
return l*b;
}
}
class example
{
public static void main(String args[])
```

```
{
cir c=new cir(3);
rec r=new rec(3,5);
//cir.display();
System.out.println("Area of circle= "+c.area());
//rec.display();
System.out.println("Area of rectangle= "+r.area());
}
}
```

Output

```
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> javac example.java
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> java example
Area of circle= 28.259999999999998
Area of rectangle= 15.0
```

 **Caution:** When extending an abstract class that contains abstract methods, you are required to either provide implementations for all the abstract methods in the subclass or declare the subclass as abstract itself.

7.2 Using Interfaces

An interface is a reference type that defines a set of abstract methods along with constants (static final variables). An interface can be considered a contract specifying the methods that a class implementing the interface must provide. Here are some key points about interfaces in Java:

Declaration: To declare an interface, you use the interface keyword followed by the interface name. Interface methods are declared without a method body.

Syntax

```
interface MyInterface {
   void myMethod(); // Abstract method declaration
}
```

Implementation: A class can implement an interface by providing implementations for all the abstract methods declared in the interface. Use the implements keyword to implement an interface.

Syntax

```
class MyClass implements MyInterface {
   public void myMethod() {
      // Method body
   }
}
```

Multiple Inheritance: Unlike classes, Java allows a class to implement multiple interfaces. This enables a form of multiple inheritance of behavior.

Syntax

```
interface Interface1 {
   void method1();
}
```

```
interface Interface2 {
   void method2();
}
class MyClass implements Interface1, Interface2 {
   public void method1() {
      // Method body
   }

   public void method2() {
      // Method body
   }
}
```

Constants: Interfaces can contain constants, which are implicitly public, static, and final. These constants must be initialized.

Syntax

```
interface Constants {
   int MAX_SIZE = 10; // Constant declaration
}
```

Default Methods: Java 8 introduced default methods in interfaces, allowing interfaces to have concrete method implementations. Default methods enable backward compatibility by allowing interfaces to evolve without breaking existing implementations.

Syntax

```
interface MyInterface {
   void myMethod(); // Abstract method

   default void defaultMethod() {
      // Default method implementation
   }
}
```

Static Methods: Java 8 also introduced static methods in interfaces. Static methods can be defined in interfaces and called using the interface name.

Syntax

```
interface MyInterface {
   static void staticMethod() {
      // Static method implementation
   }
}
```

Functional Interfaces: A functional interface is an interface with a single abstract method. Functional interfaces can be used as lambda expressions, enabling functional programming in Java.

 Example

```
@FunctionalInterface
interface MyFunctionalInterface {
   void myMethod();
}
```

Interfaces play a crucial role in Java programming, providing a mechanism for defining contracts, achieving abstraction, and enabling polymorphism. They are widely used in Java APIs, frameworks, and libraries for defining specifications and providing flexibility in implementation.

Implementation of Abstract Class Using Interface

In Java, you cannot directly implement an abstract class using an interface, as they are two different concepts. Abstract classes can provide both abstract and concrete methods, while interfaces can only contain method signatures without method bodies. However, you can achieve similar functionality by defining an interface with methods and then providing concrete implementations of those methods in a class.

Here's an example illustrating how you can implement a similar concept using an interface:

 Example

```
// Implementing abstract class using interfaces
interface I1
{
void show();
void display();
void done();
}
abstract class stu implements I1
{
public void show()
{
System.out.println("Implementing show using interface");
}
public void display()
{
System.out.println("Implementing display using interface");
}
}
class s1 extends stu
{
public void done()
{
System.out.println("Implementation of abstract class using interface is done");
}
}
class a1
```

```
{
public static void main(String args[])
{
s1 o=new s1();
o.show();
o.display();
o.done();
}
}
```

Output

```
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> javac a1.java
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> java a1
Implementing show using interface
Implementing display using interface
Implementation of abstract class using interface is done
```

Implementation of Multiple Inheritance Using Interfaces

We can achieve a form of multiple inheritance by using interfaces in java. While Java does not support multiple inheritance of classes (i.e., a class cannot directly extend more than one class), it allows a class to implement multiple interfaces. By doing so, a class can inherit behavior from multiple sources. Here's an example illustrating multiple inheritance using interfaces:

 Example

```
// Implementing multiple inheritance using interfaces
// Implementation of multiple inheritance
interface mulin
{
void show();
}
interface intr1
{
void display();
}
class stu implements mulin,intr1
{
public void show()
{
System.out.println("Implementing mutiple inheritance using I1 ");
}
public void display()
{
System.out.println("Implementing Multiple inheritance using interface 2");
```

```
}
}
class MI
{
public static void main(String args[])
{
stu s=new stu();
s.show();
s.display();
}
}
```

Output

```
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> javac MI.java
PS C:\Users\Harjinder Kaur\OneDrive\Desktop> java MI
Implementing mutiple inheritance using I1
Implementing Multiple inheritance using interface 2
```

 **Note**: In Java, when defining methods within an interface, the compiler automatically adds the public and abstract keywords before each method declaration. Additionally, for data members defined within an interface, the compiler adds the public, static, and final keywords to them.

Benefits of Interfaces

Contractual Agreement: Interfaces provide a contract for all implementing classes, ensuring that they adhere to a specific set of methods. This promotes consistency and ensures that classes fulfill their designated roles.

Abstraction and Hierarchy: Interfaces allow developers to define types and establish top-level hierarchies in their code. By programming interfaces rather than concrete implementations, developers can create flexible and extensible systems.

Multiple Inheritance: Java classes can implement multiple interfaces, enabling them to inherit behavior from multiple sources. This promotes code reuse and modularity by allowing classes to fulfill multiple roles simultaneously.

Drawbacks of Interfaces

Rigidity in Method Definitions: Once defined, interface methods cannot be modified without breaking compatibility with implementing classes. This requires careful consideration during the design phase to avoid introducing breaking changes later on.

Interface Proliferation: In complex systems, the use of interfaces may lead to a proliferation of interface definitions. This can make the codebase harder to understand and maintain, particularly if interfaces extend other interfaces in a deep hierarchy.

Limitations on Accessing Additional Methods: Since interface types do not include methods beyond those defined in the interface itself, implementing classes cannot expose additional methods directly through the interface type. This may require casting to the implementing class type to access additional functionality.

Summary

- An abstract class is a class that cannot be instantiated on its own and may contain abstract methods, which are methods without a body.
- Abstract classes are designed to serve as base classes for other classes to inherit from. They define common behavior and characteristics that subclasses can share.
- To declare an abstract class, use the abstract keyword before the class name. Abstract methods are declared with the abstract keyword and do not contain a method body.
- Abstract classes cannot be instantiated directly using the new keyword. They can only be used as base classes for other classes.
- Abstract classes can have abstract methods, which are declared without a body. Subclasses must provide implementations for these methods.
- Abstract classes can also contain concrete methods, which have a body. Subclasses inherit these concrete methods along with the abstract methods.
- Abstract classes can have a mix of abstract and concrete methods, allowing for partial abstraction and providing default behavior that subclasses can override if needed.
- Inheritance is a fundamental feature of object-oriented programming in Java that allows a class (subclass or child class) to inherit properties and behaviors from another class (superclass or parent class).
- An interface in Java is a reference type that specifies a set of abstract methods along with constants. It acts as a contract that classes must adhere to by implementing all of its methods.
- Interfaces are declared using the interface keyword followed by the interface name and the body containing method signatures.
- A class can implement multiple interfaces, allowing it to inherit behavior from multiple sources.
- Interfaces can contain constants, which are implicitly public, static, and final. They can be accessed using the interface name.

Keywords

abstract: It is used to declare an abstract class or method.

implements: Used to indicate that a class is implementing an interface.

extends: It is used to indicate that a class is inheriting from another class.

super: Used to call a superclass constructor or method from within a subclass.

interface: It is used to declare an interface, which defines a contract specifying a set of methods that implementing classes must provide.

static: It is used to declare static methods in an interface.

Self Assessment

1. What keyword is used to declare an abstract class in Java?

A. class
B. abstract
C. interface
D. implements

2. Which of the following statements about abstract classes is true?
A. Abstract classes cannot contain any concrete methods.
B. Abstract classes must be declared as public.
C. Abstract classes cannot be extended by other classes.
D. Abstract classes can contain both abstract and concrete methods.

3. Which of the following is not allowed in an abstract class?
A. Declaring a constructor
B. Declaring static methods
C. Declaring final methods
D. Declaring private methods

4. What happens if a subclass fails to implement all abstract methods from its abstract superclass?
A. The subclass will inherit the abstract methods from the superclass.
B. The subclass will become an abstract class itself.
C. The program will compile successfully.
D. The program will throw a runtime exception.

5. Which of the following is a valid statement regarding abstract classes?
A. Abstract classes can be instantiated using the new keyword.
B. Abstract classes cannot contain any fields.
C. Abstract classes can extend multiple other abstract classes.
D. Abstract methods must have a method body.

6. Which keyword is used to declare an interface in Java?
A. interface
B. class
C. abstract
D. implements

7. What is the primary purpose of interfaces in Java?
A. To provide multiple inheritance of implementation
B. To define a blueprint for classes
C. To enforce encapsulation
D. To allow private method declarations

8. In Java, a class can implement _____ interfaces at the same time.
A. One
B. Multiple
C. Two
D. No

9. Which of the following is not allowed in an interface?
A. Declaring constants
B. Declaring abstract methods

C. Declaring constructors
D. Declaring default methods

10. Java 8 introduced which feature for interfaces?
A. Static methods
B. Final methods
C. Private methods
D. Protected methods

11. Which keyword is used to implement multiple interfaces in a Java class?
A. extends
B. inherits
C. implements
D. extends and implements

12. In Java, why does multiple inheritance through classes not exist?
A. To avoid the diamond problem
B. Java does not support inheritance
C. To simplify the language
D. Because it leads to performance issues

13. How can multiple inheritance be achieved in Java using interfaces?
A. By extending multiple interfaces using the 'extends' keyword
B. By declaring multiple interface variables in the class
C. By implementing multiple interfaces separated by commas using the 'implements' keyword
D. By using nested interfaces

14. Which of the following statements is true about interfaces in Java?
A. Interfaces can only contain abstract methods
B. Interfaces cannot extend other interfaces
C. Interfaces cannot be implemented by classes
D. Interfaces can only contain static methods

15. Which of the following is a benefit of using interfaces for multiple inheritance?
A. It allows sharing of implementation code between classes
B. It helps avoid conflicts that arise with multiple inheritance through classes
C. It improves runtime performance
D. It simplifies the code structure

Answers for Self Assessment

1.	B	2.	D	3.	D	4.	B	5.	D
6.	A	7.	B	8.	B	9.	C	10.	A
11.	C	12.	A	13.	C	14.	B	15.	B

Review Questions

1. Explain the concept and rules of creating abstract classes in Java. What is their purpose, and how are they used in object-oriented programming?
2. Describe the role of abstract methods within abstract classes. How are abstract methods declared, explain with the help of an example ?
3. Explain the relationship between abstract classes and concrete subclasses. How do subclasses inherit from abstract classes, and what are the requirements for subclassing?
4. Explain how abstract classes contribute to achieving abstraction and code reusability in Java. Provide examples to illustrate these concepts.
5. What is an interface in Java, and what role does it play in object-oriented programming? Explain the purpose of interfaces and how they contribute to achieving abstraction and polymorphism in Java.
6. Discuss the syntax and rules for defining interfaces in Java. How are interfaces used for the abstract classes?
7. Explain how interfaces are implemented in Java classes. Describe the process of implementing interfaces and providing concrete implementations for interface methods.
8. Compare and contrast abstract classes with interfaces in Java. What are the similarities and differences between the two, and when would you choose one over the other?

Further Readings

Horstmann, C. S., & Cornell, G. (2008). *Core Java: Advanced Features* (Vol. 2). Prentice Hall.

Bloch, J. (2008). *Effective java (the java series)*. Prentice Hall PTR.

Choudhary, H. H., & Warth, C. J. (2013). Core Java Professional.

Nayak, P. (2021). *Core Java-The Practical Guide For Beginners*. Pravuram Nayak.

Das, R. K. (2011). CORE JAVA for Beginners: Revised Edition.

Flanagan, D. (2005). *Java in a Nutshell*. " O'Reilly Media, Inc.".

Web Links

https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

https://www.scaler.com/topics/java/abstract-class-in-java/

https://www.baeldung.com/java-abstract-class

https://www.geeksforgeeks.org/abstract-classes-in-java/

https://www.digitalocean.com/community/tutorials/interface-in-java

https://www.codecademy.com/resources/docs/java/interfaces

https://dspmuranchi.ac.in/pdf/Blog/interface%20and%20abstract%20class.pdf

Unit 08: Nested Classes

CONTENTS

Objectives

- Understand the concept of nested classes and their role in organizing code.
- Differentiate between static nested classes and inner classes.
- Recognize the purpose of wrapper classes in Java.
- Understand how wrapper classes convert primitive data types into objects.

Introduction

Nested classes offer a way to organize code more effectively by allowing classes to be defined within the scope of another class. This feature enhances encapsulation and modularity, as it enables the grouping of related classes together. Static nested classes serve to logically associate utility classes or constants with the enclosing class, while inner classes provide a means to encapsulate functionality closely tied to instances of the enclosing class.

Static nested classes, being associated with the enclosing class itself, do not have access to the instance variables of the enclosing class directly. They are often utilized for providing helper functionality or organizing related classes within a single namespace. On the other hand, inner classes, including member, local, and anonymous inner classes, can access both static and instance members of the enclosing class. They are commonly used to implement more complex behavior within the context of the enclosing class, enhancing encapsulation and readability.

Overall, nested classes facilitate better code organization and encapsulation by allowing related classes to be nested within one another. This hierarchical structure aids in improving code readability and maintainability, as it makes the relationships between classes more explicit and intuitive. Additionally, nested classes help to reduce namespace pollution by encapsulating related functionality within a single class or scope.

8.1 Inner Classes in Java with Examples

In this article, I am going to discuss Inner Classes in Java with Examples. Please read our previous article, where we discussed **Constructors in Java**. At the end of this article, you will understand what are inner classes and their type and when and how to implement this in Java Applications.

What are Inner Classes in Java?

The inner class is defined inside the body of another class (known as an outer class). The class written within is called the nested class, and the class that holds the inner class is called the outer class. Java inner class can be declared private, public, protected, or with default access whereas an outer class can have only public or default access.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable. The Syntax is given below.

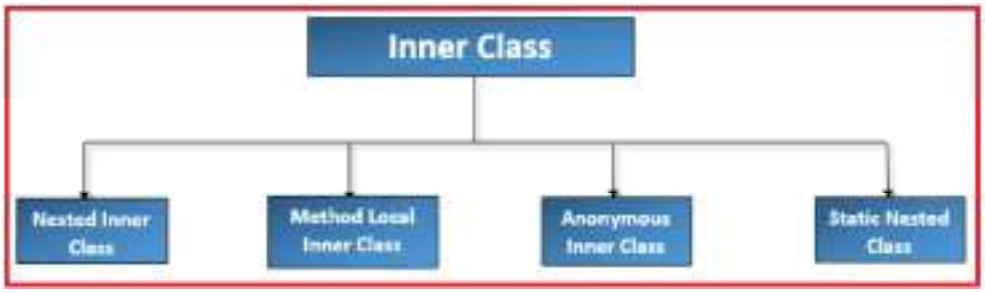
```
class Outer_class {
        class Inner_class {
                //code
        }
}
```

8.2 Types of Inner Classes in Java

The inner classes are of four types.

They are :

- **Nested Inner Class**
- **Method Local Inner Class**
- **Anonymous Inner Class**
- **Static Nested Class**



Nested Inner Class in Java:

A class created within the class and outside the method is known as Nested Inner Class in Java. It can access the private instance variable of the outer class.

Example to Understand Nested Inner Classes in Java:

```
package Demo;
public class NestedInnerClass
{
class Inner
{
public void show ()
{
System.out.println ("In a nested class method");
}
}
public static void main (String[]args)
{
NestedInnerClass.Inner in = new NestedInnerClass ().new Inner ();
in.show ();
}
}
```

Output: In a nested class method

8.3 Method Local Inner Class in Java

A class created within the method of the enclosing class is known as Method Local Inner Class. Since the local inner class is not associated with Object, we can't use private, public, or protected access modifiers with it. The only allowed modifiers are abstract or final.

Method Local Inner Class Example in Java:

```
package Demo;
public class MethodLocalInnerClass
{
void outerMethod ()
{
System.out.println ("Inside OuterMethod");
// Inner class is local to outerMethod()
class Inner
{
void innerMethod ()
{
System.out.println ("Inside InnerMethod");
}
}
Inner y = new Inner ();
y.innerMethod ();
}
```

```
public static void main (String[]args)
{
MethodLocalInnerClass outer = new MethodLocalInnerClass ();
outer.outerMethod ();
}
}
```

Output:

Inside OuterMethod

Inside InnerMethod

8.4 Anonymous Inner Class in Java

An inner class declared without a class name is known as an anonymous inner class in Java. It is created for implementing an interface or extending a class. Since an anonymous class has no name, it is not possible to define a constructor for an anonymous class. Its name is decided by the java compiler.

Example to Understand Anonymous Inner Class in Java:

```
package Demo;
abstract class Animal
{
abstract void dog ();
}
class AnonymousInnerClass
{
public static void main (String args[])
{
Animal p = new Animal ()
{
void dog ()
{
System.out.println ("Dog is an Animal.");
}
};
p.dog ();
}
}
```

Output: Dog is an Animal.

8.5 Static Nested Class in Java

Static nested classes are not technically inner classes. They are like static members of the outer class. A static nested class is the same as any other top-level class and is nested for only packaging convenience. Because this is static in nature so this type of inner class doesn't share any special kind

of relationship with an instance of the outer class. A static nested class cannot access non-static members of the outer class.

Example to Understand Static Nested Class in Java:

```
package Demo;
public class StaticNestedClass
{
static class Nested_Demo
{
public void my_method ()
{
System.out.println ("This is my nested class");
}
}
public static void main (String args[])
{
StaticNestedClass.Nested_Demo nested = new StaticNestedClass.Nested_Demo ();
nested.my_method ();
}
}
```

Output: This is my nested class

8.6 Advantages of Inner Classes in Java

- It helps in Code Optimization.
- If a class is useful to only one class, it makes sense to keep it nested and together. It helps in the packaging of the classes.
- It has nested classes that are used to develop a more readable and maintainable code.
- The inner classes can access outer class private members and at the same time, we can hide the inner class from the outer world.
- It requires less code to write.

Java inner class or nested class is a class that is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.

Additionally, it can access all the members of the outer class, including private data members and methods.

Syntax of Inner class

```
1. class Java_Outer_class{
2.  //code
3.  class Java_Inner_class{
4.   //code
5.  }
6. }
```

Advantage of Java inner classes

There are three advantages of inner classes in Java. They are as follows:

1. Nested classes represent a particular type of relationship that is **it can access all the members (data members and methods) of the outer class,** including private.
2. Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
3. **Code Optimization**: It requires less code to write.

Need of Java Inner class

Sometimes users need to program a class in such a way so that no other class can access it. Therefore, it would be better if you include it within other classes.

If all the class objects are a part of the outer object then it is easier to nest that class inside the outer class. That way all the outer class can access all the objects of the inner class.

Do You Know

- What is the internal code generated by the compiler for member inner class?
- What are the two ways to create an anonymous inner class?
- Can we access the non-final local variable inside the local inner class?
- How to access the static nested class?
- Can we define an interface within the class?
- Can we define a class within the interface?

8.7 Difference between nested class and inner class in Java

An inner class is a part of a nested class. Non-static nested classes are known as inner classes.

Types of Nested classes

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

Non-static nested class (inner class)

1. Member inner class
2. Anonymous inner class
3. Local inner class

Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing an interface or extending class. The java compiler decides its name.
Local Inner Class	A class was created within the method.
Static Nested Class	A static class was created within the class.

Nested Interface	An interface created within class or interface.

8.8 Wrapper classes in Java

The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive*.

Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float

double	Double

Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

Wrapper class Example: Primitive to Wrapper

```
//Java program to convert primitive into objects
//Autoboxing example of int to Integer
public class WrapperExample1{
public static void main(String args[]){
//Converting int into Integer
int a=20;
Integer i=Integer.valueOf(a);//converting int into Integer explicitly
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
System.out.println(a+" "+i+" "+j);
}}
```

Output:

20 20 20

8.9 Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

Wrapper class Example: Wrapper to Primitive

```
//Java program to convert object into primitives
//Unboxing example of Integer to int
public class WrapperExample2{
public static void main(String args[]){
//Converting Integer to int
Integer a=new Integer(3);
int i=a.intValue();//converting Integer to int explicitly
int j=a;//unboxing, now compiler will write a.intValue() internally

System.out.println(a+" "+i+" "+j);
}}
```

Output:

3 3 3

8.10 Java Wrapper classes Example

```
//Java Program to convert all primitives into its corresponding
//wrapper objects and vice-versa
public class WrapperExample3{
public static void main(String args[]){
byte b=10;
short s=20;
int i=30;
long l=40;
float f=50.0F;
double d=60.0D;
char c='a';
boolean b2=true;
```

//Autoboxing: Converting primitives into objects

```
Byte byteobj=b;
Short shortobj=s;
Integer intobj=i;
Long longobj=l;
Float floatobj=f;
Double doubleobj=d;
Character charobj=c;
Boolean boolobj=b2;

//Printing objects
System.out.println("---Printing object values---");
System.out.println("Byte object: "+byteobj);
System.out.println("Short object: "+shortobj);
System.out.println("Integer object: "+intobj);
System.out.println("Long object: "+longobj);
System.out.println("Float object: "+floatobj);
System.out.println("Double object: "+doubleobj);
System.out.println("Character object: "+charobj);
System.out.println("Boolean object: "+boolobj);

//Unboxing: Converting Objects to Primitives
byte bytevalue=byteobj;
short shortvalue=shortobj;
int intvalue=intobj;
```

```
    long longvalue=longobj;
    float floatvalue=floatobj;
    double doublevalue=doubleobj;
    char charvalue=charobj;
    boolean boolvalue=boolobj;

    //Printing primitives
    System.out.println("---Printing primitive values---");
    System.out.println("byte value: "+bytevalue);
    System.out.println("short value: "+shortvalue);
    System.out.println("int value: "+intvalue);
    System.out.println("long value: "+longvalue);
    System.out.println("float value: "+floatvalue);
    System.out.println("double value: "+doublevalue);
    System.out.println("char value: "+charvalue);
    System.out.println("boolean value: "+boolvalue);
    }}
```

Output:

---Printing object values---

Byte object: 10

Short object: 20

Integer object: 30

Long object: 40

Float object: 50.0

Double object: 60.0

Character object: a

Boolean object: true

---Printing primitive values---

byte value: 10

short value: 20

int value: 30

long value: 40

float value: 50.0

double value: 60.0

char value: a

boolean value: true

8.11 Custom Wrapper class in Java

Java Wrapper classes wrap the primitive data types, that is why it is known as wrapper classes. We can also create a class which wraps a primitive data type. So, we can create a custom wrapper class in Java.

```
//Creating the custom wrapper class
class Javatpoint{
private int i;
Javatpoint(){}
Javatpoint(int i){
this.i=i;
}
public int getValue(){
return i;
}
public void setValue(int i){
this.i=i;
}
@Override
public String toString() {
  return Integer.toString(i);
}
}
//Testing the custom wrapper class
public class TestJavatpoint{
public static void main(String[] args){
Javatpoint j=new Javatpoint(10);
System.out.println(j);
}}
```

Output:

10

Summary

- Nested classes, inner classes, and wrapper classes are all features in object-oriented programming languages like Java, offering distinct functionalities and benefits.
- Nested classes enable the definition of a class within another class, aiding in code organization and encapsulation.
- Inner classes, a specific type of nested class, provide access to the members of the enclosing class, including private ones, and are commonly used for implementing complex behavior within the context of the enclosing class.
- They come in various forms such as member, local, and anonymous inner classes, each serving different purposes like enhancing modularity and improving readability.
- Wrapper classes, on the other hand, are used to convert primitive data types into objects, allowing them to be used in scenarios where objects are required, such as collections.
- They provide utility methods for converting between primitive data types and their corresponding object types, as well as additional functionality like parsing and comparison.

- Wrapper classes help bridge the gap between primitive types and objects, enabling a more unified approach to data manipulation and manipulation in object-oriented programming environments. Overall, nested classes, inner classes, and wrapper classes each play unique roles in enhancing code structure, encapsulation, and data manipulation capabilities in object-oriented programming languages.

Keywords

Nested Classes: Nested classes are classes defined within the scope of another class, allowing for a more organized code structure and improved encapsulation. They can be static or non-static and provide a way to logically group related classes together.

Inner Classes: Inner classes are a specific type of nested class that have access to the members of the enclosing class, including private ones. They are primarily used for implementing complex behavior within the context of the enclosing class and come in various forms such as member, local, and anonymous inner classes.

Wrapper Classes: Wrapper classes, also known as boxed types, are classes that encapsulate primitive data types in object form. They provide a way to treat primitive data types as objects, allowing for additional functionality such as methods and compatibility with collections that require objects rather than primitives. Examples include Integer, Double, and Boolean in Java.

Static Nested Classes: Static nested classes are nested classes that are associated with the enclosing class itself rather than with instances of the enclosing class. They can access only static members of the enclosing class directly and are commonly used for grouping related utility classes or constants.

Member Inner Classes: Member inner classes are inner classes that are defined at the member level of the enclosing class. They have access to both static and non-static members of the enclosing class and are instantiated with an instance of the enclosing class.

Local Inner Classes: Local inner classes are inner classes that are defined within a block of code, typically within a method. They have access to the variables of the enclosing block, including local variables, and are often used for implementing complex logic within a limited scope.

Anonymous Inner Classes: Anonymous inner classes are inner classes without a name that are defined and instantiated in a single expression. They are commonly used for implementing interfaces or extending classes in-line, particularly for event handling or callback mechanisms.

Self Assessment

1. What is a nested class?

A. A class defined within another class
B. A class with only static members
C. A class used for wrapping primitive data types
D. A class that inherits from another class

2. Which type of inner class has access to both static and non-static members of the enclosing class?

A. Static inner class
B. Member inner class
C. Local inner class
D. Anonymous inner class

3. Which of the following is an example of a wrapper class in Java?

A. String

B. Integer
C. StringBuilder
D. ArrayList

4. What is the purpose of a static nested class?

A. To encapsulate behavior within an instance of the enclosing class
B. To access only static members of the enclosing class
C. To provide additional functionality for primitive data types
D. To implement interfaces in-line

5. Which type of inner class is defined within a block of code, typically within a method?

A. Static inner class
B. Member inner class
C. Local inner class
D. Anonymous inner class

6. In Java, which keyword is used to define an inner class?

A. class
B. outer
C. nested
D. inner

7. Which of the following statements about inner classes is true?

A. Inner classes can only access public members of the enclosing class.
B. Inner classes cannot be instantiated.
C. Inner classes can access private members of the enclosing class.
D. Inner classes cannot have constructors.

8. Which of the following is NOT a wrapper class in Java?

A. Integer
B. Character
C. Float
D. Math

9. What is the main advantage of using inner classes in Java?

A. Improved encapsulation
B. Reduced code complexity
C. Enhanced performance
D. Easier debugging

10. Which type of inner class is defined without a name?

A. Static inner class

B. Member inner class
C. Local inner class
D. Anonymous inner class

11. Which of the following statements about static nested classes is true?

A. They have access to non-static members of the enclosing class.
B. They are always instantiated with an instance of the enclosing class.
C. They are defined within a method.
D. They can be declared with the abstract keyword.

12. What is the purpose of a wrapper class in Java?

A. To provide additional security for sensitive data
B. To improve memory efficiency
C. To convert primitive data types into objects
D. To restrict access to class members

13. Which type of inner class is associated with instances of the enclosing class?

A. Static inner class
B. Member inner class
C. Local inner class
D. Anonymous inner class

14. In Java, wrapper classes are used primarily for:

A. Controlling access to class members
B. Enhancing performance
C. Converting primitive data types into objects
D. Defining nested classes

15. Which of the following is NOT a characteristic of an inner class in Java?

A. Can access private members of the enclosing class
B. Can be declared static
C. Can be defined within a method
D. Can have its own constructors

Answers for Self Assessment

1.	A	2.	B	3.	B	4.	B	5.	C
6.	A	7.	C	8.	D	9.	A	10.	D
11.	A	12.	C	13.	B	14.	C	15.	B

Review Questions

1) Explain the concept of nested classes and provide a scenario where they might be useful in a software application.
2) Describe the difference between static nested classes and inner classes in Java. Provide examples for each.
3) Discuss the advantages and disadvantages of using wrapper classes in Java. Provide examples of situations where wrapper classes are commonly used.
4) Illustrate the usage of inner classes in event handling scenarios in Java, providing a step-by-step explanation of how they are implemented.
5) Compare and contrast member inner classes, local inner classes, and anonymous inner classes in Java. Explain when each type would be most appropriate to use in a software project.

Further Readings

Balagurusamy E. Programming with Java 3e Primer. New Delhi: Tata McGraw Publishers.

Schildt H. Java A Beginner's Guide, 3rd ed. New York: Mc-Graw Hill.

Arnold, K., Gosling, J., & Holmes, D. (2005). *The Java programming language*. Addison Wesley Professional.

Haggar, P. (2000). *Practical Java: programming language guide*. Addison-Wesley Professional. Online

Web Links

https://www.tutorialsfreak.com/java-tutorial/java-inner-class

https://www.techguruspeaks.com/java-nested-class/

Unit 09: Packages

CONTENTS

Objectives

- Students will understand the concept of Java packages and their role in organizing and structuring code in large-scale software projects.
- Students will be able to create and manage user-defined packages in Java, including naming conventions and best practices for package organization.
- Students will comprehend the importance of package visibility and access modifiers in Java, and how they affect the encapsulation and accessibility of classes and members.

Introduction

In Java, packages are a mechanism for organizing and grouping related classes and interfaces. Built-in packages in Java are predefined collections of classes and interfaces that provide various functionalities, such as input/output operations, networking, graphical user interface (GUI) components, and more. Examples of built-in packages include java.util for utility classes, java.io for input/output operations, java.net for networking, and java.awt for GUI components. These packages offer a rich set of functionalities that developers can leverage to build robust and feature-rich applications without having to reinvent the wheel. By organizing classes and interfaces into packages, Java promotes code reusability, maintainability, and scalability, allowing developers to manage complex projects more efficiently.

On the other hand, user-defined packages in Java are custom packages created by developers to organize their own classes and interfaces. These packages help in structuring code logically and enhancing its readability and maintainability. To create a user-defined package in Java, developers simply need to place their classes and interfaces within a directory structure that matches the package name. By using user-defined packages, developers can encapsulate related functionality,

manage dependencies, and share code across multiple projects easily. Additionally, user-defined packages facilitate collaboration among team members by providing a clear and standardized way to organize and access code. Overall, both built-in and user-defined packages play a crucial role in Java development, enabling developers to write clean, modular, and scalable code.

9.1 Java Packages

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

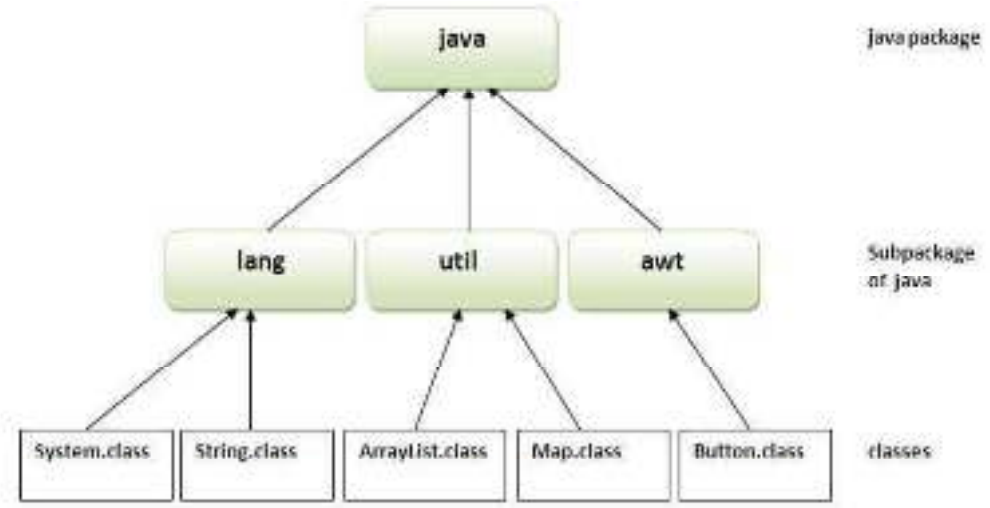
Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



Simple example of java package

The **package keyword** is used to create a package in java.

```
1. //save as Simple.java
2. package mypack;
3. public class Simple{
4.  public static void main(String args[]){
5.     System.out.println("Welcome to package");
6.   }
7. }
```

9.2 How to Compile Java Package

If you are not using any IDE, you need to follow the **syntax** given below:

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

9.3 How to Run Java Package Program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output:Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another 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) Using packagename.*

If you use 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.*

```
1.  //save by A.java
2.  package pack;
3.  public class A{
4.    public void msg(){System.out.println("Hello");}
5.  }
```

```
1.  //save by B.java
2.  package mypack;
3.  import pack.*;
4.
5.  class B{
6.    public static void main(String args[]){
7.     A obj = new A();
8.     obj.msg();
9.    }
```

```
10. }
```

Output:Hello

2) Using packagename.classname

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

Example of package by import package.classname

```
1.  //save by A.java
2.
3.  package pack;
4.  public class A{
5.    public void msg(){System.out.println("Hello");}
6.  }
1.  //save by B.java
2.  package mypack;
3.  import pack.A;
4.
5.  class B{
6.    public static void main(String args[]){
7.     A obj = new A();
8.     obj.msg();
9.    }
10. }
```

Output:Hello

3) Using fully qualified name

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

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

 Example of package by import fully qualified name

```
1.  //save by A.java
2.  package pack;
3.  public class A{
4.    public void msg(){System.out.println("Hello");}
5.  }
1.  //save by B.java
2.  package mypack;
3.  class B{
4.    public static void main(String args[]){
```

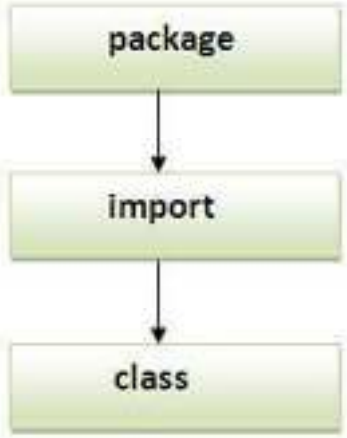
```
5.      pack.A obj = new pack.A();//using fully qualified name
6.      obj.msg();
7.     }
8.   }
```

Output:Hello

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



9.4 Subpackage in Java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

Example of Subpackage

```
1.  package com.javatpoint.core;
2.  class Simple{
3.    public static void main(String args[]){
4.     System.out.println("Hello subpackage");
5.    }
6.  }
```

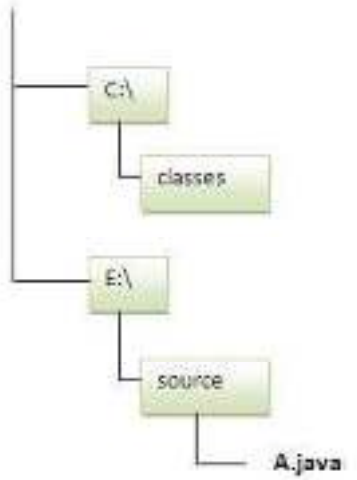
To Compile: javac -d . Simple.java

To Run: java com.javatpoint.core.Simple

Output:Hello subpackage

9.5 How to Send the Class File to Another Directory or Drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:




```
1. //save as Simple.java
2. package mypack;
3. public class Simple{
4.  public static void main(String args[]){
5.    System.out.println("Welcome to package");
6.   }
7. }
```

To Compile:

e:\sources> javac -d c:\classes Simple.java

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file reside

e:\sources> set classpath=c:\classes;.;

e:\sources> java mypack.Simple

Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

e:\sources> java -classpath c:\classes mypack.Simple

Output:Welcome to package

Ways to load the class files or jar files

There are two ways to load the class files temporary and permanent.

- Temporary
  - By setting the classpath in the command prompt
  - By -classpath switch
- Permanent
  - By setting the classpath in the environment variables
  - By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

Rule: There can be only one public class in a java source file and it must be saved by the public class name.

```
1. //save as C.java otherwise Compilte Time Error
2.
3. class A{}
4. class B{}
5. public class C{}
```

9.6 How to put two Public Classes in a Package?

If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:

```
1. //save as A.java
2.
3. package javatpoint;
4. public class A{}
1. //save as B.java
2.
3. package javatpoint;
4. public class B{}
```

User-Defined Packages in Java

Java, being an object-oriented programming language, encourages the use of modular code to improve maintainability and reusability. One of the key features that facilitate code organization is the concept of packages. Packages in Java serve as containers for related classes, interfaces, and other resources, allowing developers to structure their code in a logical and efficient manner. While Java provides a set of predefined packages, it also allows developers to create their own user-defined packages to further organize their code. In this article, we will explore the concept of user-defined packages in Java, their benefits, and how to create and utilize them effectively.

9.7 What are Packages in Java?

A package in Java is a mechanism for organizing related classes, interfaces, and resources into a single unit. It provides a hierarchical structure to the codebase, which aids in better code

management and avoids naming conflicts. Packages are represented by directories in the file system, where each directory corresponds to a package name.

Benefits of User-Defined Packages

- **Code Organization:** User-defined packages allow developers to group related classes and resources together, making it easier to navigate and understand the codebase. This organization promotes code modularity and improves overall maintainability.
- **Encapsulation:** Packages provide a level of encapsulation by allowing you to specify the access level of classes and resources within the package. This helps in controlling the visibility and accessibility of code elements, making it easier to define the boundaries of your code.
- **Code Reusability:** By creating user-defined packages, you can encapsulate commonly used classes, utilities, or modules, making them easily reusable across different projects. This reusability saves development time and effort, as well as promotes consistency across applications.

Creating User-Defined Packages

To create a user-defined package in Java, follow these steps:

Choose a meaningful package name that represents the purpose or functionality of the code it will contain. Conventionally, package names are written in lowercase and follow the reverse domain name notation, such as "com.example.mypackage."

Include the package declaration as the first line of each Java source file that belongs to the package. For example, if you choose the package name "com.example.mypackage," the package declaration should be: package com.example.mypackage;

Place the Java source files within a directory structure that reflects the package hierarchy. For example, if the package is "com.example.mypackage," the source file should be located at "com/example/mypackage/MyClass.java" in the file system.

Here's an example of how to create and use a user-defined package in Java:

Let's say we want to create a package named "com.example" and include a class called "**Calculator**" within it.

Create a new directory structure for your package. In your project folder, create a folder named "com" and within it, create another folder named "example".

Create the "**Calculator.java**" file and place it inside the "com/example" folder.

com/example/Calculator.java

```
1.  package com.example;
2.  public class Calculator {
3.      public int add(int a, int b) {
4.          return a + b;
5.      }
6.      public int subtract(int a, int b) {
7.          return a - b;
8.      }
9.      public int multiply(int a, int b) {
10.         return a * b;
11.     }
12.     public int divide(int a, int b) {
13.         if (b != 0) {
```

```
14.             return a / b;
15.         } else {
16.             throw new ArithmeticException("Cannot divide by zero!");
17.         }
18.     }
19. }
```

Now, create another file outside the "com" folder to access the Calculator class from the user-defined package.

PackageExample.java

```
1.  import com.example.Calculator;
2.  public class PackageExample {
3.      public static void main(String[] args) {
4.          Calculator calculator = new Calculator();
5.          int result = calculator.add(5, 3);
6.          System.out.println("Addition: " + result);
7.          result = calculator.subtract(5, 3);
8.          System.out.println("Subtraction: " + result);
9.          result = calculator.multiply(5, 3);
10.         System.out.println("Multiplication: " + result);
11.         result = calculator.divide(10, 2);
12.         System.out.println("Division: " + result);
13.     }
14. }
```

Output:

Addition: 8

Subtraction: 2

Multiplication: 15

Division: 5

When we compile and run the program, it will access the Calculator class from the "com.example" package. The code performs basic mathematical operations using the methods of the Calculator class and displays the results.

Remember to compile both files together using the command: **javac PackageExample.java.**

After successful compilation, we can run the code using the command: **java PackageExample.**

This example demonstrates how to create and use a user-defined package in Java. The package allows you to organize and encapsulate related classes together, making your code more modular and maintainable.

9.8 Utilizing User-Defined Packages

Once you have created a user-defined package, you can use its classes and resources in other Java files or projects. Here's how you can utilize user-defined packages:

- **Importing Packages:** To use classes from a user-defined package in another Java file, import the required classes using the import statement. For example: import com.example.mypackage.MyClass;

- **Access Control:** By default, classes and resources within a package have package-private access, which means they can only be accessed by other classes within the same package. If you want to provide access to classes or resources outside the package, use appropriate access modifiers (e.g., public, protected).
- **Packaging and Distribution:** When distributing your Java project, ensure that the package structure is maintained, and the necessary files and directories are included. This allows others to easily import and use your user-defined package in their own projects.

Summary

- In Java, packages serve as containers for organizing classes and interfaces into logical groups, providing a means of encapsulation and namespace management. Built-in packages are those provided by the Java Standard Library, covering a wide range of functionalities such as I/O operations, networking, GUI development, and more.
- These packages are essential components of Java development and are readily available for use in any Java program. Examples of built-in packages include java.lang, java.util, java.io, and java.net, among others. By utilizing these packages, developers can leverage pre-implemented functionalities, saving time and effort in coding commonly required features.
- On the other hand, user-defined packages are packages created by developers to organize their own classes and interfaces. These packages allow developers to structure their codebase in a modular and maintainable manner, enhancing code readability and reusability.
- User-defined packages help in managing large-scale projects by facilitating the separation of concerns and providing a clear hierarchy of dependencies. They also support access control through the use of access modifiers like public, private, and protected, ensuring proper encapsulation and abstraction. Overall, both built-in and user-defined packages are integral aspects of Java programming, enabling developers to build robust, scalable, and well-organized applications.

Keywords

java.lang: The java.lang package is automatically imported into every Java program and contains fundamental classes and exceptions used in basic language operations, such as Object, String, Math, and exceptions like NullPointerException.

java.util: The java.util package provides utility classes and data structures to support common programming tasks, including collections like ArrayList, HashMap, and utility classes for date and time manipulation, random number generation, and more.

java.io: The java.io package offers classes for performing input and output operations, facilitating file handling, streams, serialization, and other forms of I/O operations necessary for interacting with external resources.

java.net: The java.net package provides classes and interfaces for networking operations, enabling developers to create network applications, communicate over the internet, establish connections, handle sockets, and manage network protocols.

java.awt: The java.awt (Abstract Window Toolkit) package contains classes for creating graphical user interfaces (GUIs) in Java, offering components for building windows, buttons, text fields, menus, and other GUI elements for desktop applications.

User-defined Packages:

1. **Package**: A package in Java is a namespace that organizes a set of related classes and interfaces, providing a hierarchical structure for managing and grouping components within a project.

2. **Directory Structure**: User-defined packages are organized within the directory structure of a Java project, where each package corresponds to a directory in the file system. The directory structure mirrors the package hierarchy, facilitating code organization and navigation.
3. **Package Declaration**: A package declaration is a statement at the beginning of a Java source file that specifies the package to which the classes and interfaces defined in the file belong. It helps in categorizing and identifying the components within a project.
4. **Access Control**: User-defined packages support access control mechanisms, allowing developers to specify the visibility of classes and interfaces within the package hierarchy using access modifiers such as public, private, protected, or default (package-private).
5. **Encapsulation**: Encapsulation is a key principle in object-oriented programming (OOP) that user-defined packages facilitate. By encapsulating related classes and interfaces within packages, developers can control access to the components, hide implementation details, and promote modular design and code reusability.

Self Assessment

1. What is a Java package?

A. A folder that contains Java classes

B. A collection of related classes and interfaces

C. A compiled Java program

D. A Java archive file

2. Which keyword is used to declare a package in Java?

A. import

B. package

C. class

D. public

3. What is the default package in Java?

A. java.util

B. java.lang

C. java.io

D. There is no default package

4. Which of the following statements about importing packages in Java is correct?

A. Packages must be imported explicitly to use their classes

B. All packages are imported automatically by the compiler

C. import statements are not required in Java

D. Importing packages is optional in Java

5. Which statement is true about naming conventions for Java packages?

A. Package names must be unique across all Java projects

B. Package names must be all lowercase

C. Package names can contain spaces

D. Package names must be unique within a project

6. Which of the following statements is correct regarding the hierarchy of packages in Java?
A. Packages can contain only one sub-package
B. Packages cannot be nested
C. Packages can have multiple levels of nesting
D. Java does not support package hierarchy

7. What is the purpose of using packages in Java?
A. To organize classes and interfaces into meaningful groups
B. To decrease the accessibility of classes
C. To increase code duplication
D. To make classes inaccessible to other classes

8. Which of the following statements about the 'java.lang' package is true?
A. It needs to be imported explicitly in every Java program
B. It contains classes that are automatically imported by the compiler
C. It's not a built-in package in Java
D. It's deprecated in newer versions of Java

9. How do you access a class from a different package within the same project?
A. By using the import statement
B. By using the package statement
C. By extending the class
D. By renaming the class

10. Which of the following statements about user-defined packages in Java is true?
A. User-defined packages are not allowed in Java
B. User-defined packages must always be imported explicitly
C. User-defined packages should be stored in the 'java' directory
D. User-defined packages can be created to organize related classes

11. What happens if two packages have classes with the same name?
A. The compiler throws an error
B. The JVM throws an error
C. Both classes can be used in the same program without any issues
D. The class in the package imported last will be used

12. Which statement is true regarding the visibility modifiers in Java?
A. Classes in the same package can access package-private members
B. Private members are accessible from any class
C. Public members cannot be accessed from outside the package
D. Protected members are accessible only within the same class

13. What is the difference between 'import package.' and 'import package.ClassName'?

A. There is no difference
B. 'import package.' imports all classes from the package, while 'import package.ClassName' imports only the specified class
C. 'import package.' imports only the specified class, while 'import package.ClassName' imports all classes from the package
D. 'import package.' is not a valid import statement

14. Which of the following is a valid package declaration?
A. package main.java;
B. package com.example;
C. package 123abc;
D. package java.util.*;

15. What is the purpose of using the 'classpath' when working with packages in Java?
A. To specify the path where Java should look for packages
B. To define the package hierarchy
C. To organize classes within a package
D. To set the classpath for the JVM

Answers for Self Assessment

1.	B	2.	B	3.	B	4.	A	5.	D
6.	C	7.	A	8.	B	9.	A	10.	D
11.	D	12.	A	13.	B	14.	B	15.	A

Review Questions

1) Explain the concept of Java packages and their significance in software development.
2) Discuss the advantages of using packages in Java programming. Provide examples to support your explanation.
3) Describe the process of creating and organizing user-defined packages in Java. Include the steps involved and best practices.
4) Explain the access modifiers in Java (public, private, protected, default) and how they affect the visibility of classes and members within and outside packages.
5) Discuss the role of the classpath in Java and how it relates to package management. Provide examples of how classpath settings can impact package usage and compilation in Java programs.

Further Readings

Balagurusamy E. Programming with Java 3e Primer. New Delhi: Tata McGraw Publishers.

Schildt H. Java A Beginner's Guide, 3rd ed. New York: Mc-Graw Hill.

Arnold, K., Gosling, J., & Holmes, D. (2005). *The Java programming language*. Addison

Wesley Professional.

Haggar, P. (2000). *Practical Java: programming language guide*. Addison-Wesley Professional. Online

Web Links

http://www.javabeginner.com/learn-java/java-string-comparison

http://www.leepoint.net/notesjava/data/strings/55stringTokenizer/10stringtokenizer.html

http://admashmc.com/main/images/Lec_Notes/javaarray.pdf

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

https://www.w3schools.com/java/java_arrays.asp

https://www.baeldung.com/java-arrays-guide

Unit 10: More on Packages

CONTENTS

Objectives

- Ability to explain the concept of inheritance in Java.
- Proficiency in demonstrating inheritance through code examples.
- Knowledge of how inheritance promotes code reuse and supports the concept of polymorphism.

Introduction

In Java, packages serve as a mechanism for organizing and grouping related classes and interfaces. When creating a package, you typically define a directory structure that corresponds to the package name, and then place your Java files inside these directories. For instance, if you want to create a package named `com.example`, you would create a directory named `com` within your project directory, and within that directory, create another directory named `example`. Your Java files would then reside within the `example` directory. By importing packages, you can access classes and interfaces defined in those packages from within your own code. This is done using the `import` statement at the beginning of your Java file, followed by the package name and optionally the specific class or interface you want to import.

The Java API (Application Programming Interface) is a vast collection of classes and interfaces provided by Java that offer a wide range of functionality for developing various types of applications. These classes and interfaces are organized into packages, covering areas such as I/O operations, networking, database connectivity, graphical user interfaces (GUIs), data structures, and more. Developers can leverage the Java API to expedite development by utilizing pre-existing, well-tested classes and interfaces rather than reinventing the wheel. By importing the necessary

packages and classes from the Java API into their own code, developers can harness the power and versatility of Java to build robust, efficient, and scalable applications for a multitude of purposes.

10.1 What is Java Package?

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

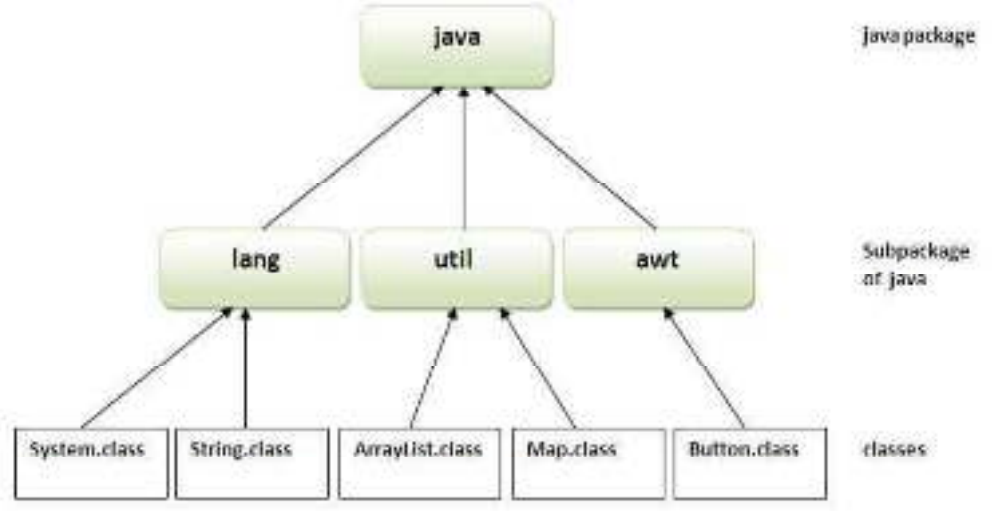
Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



Simple example of java package

The **package keyword** is used to create a package in java.

```
1. //save as Simple.java
2. package mypack;
3. public class Simple{
4.  public static void main(String args[]){
5.    System.out.println("Welcome to package");
6.  }
7. }
```

10.2 How to Compile Java Package

If you are not using any IDE, you need to follow the **syntax** given below:

1. javac -d directory javafilename

For **example**

1. **javac -d . Simple.java**

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output:Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another 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) Using packagename.*

If you use 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.*

```
1.  //save by A.java
2.  package pack;
3.  public class A{
4.    public void msg(){System.out.println("Hello");}
5.  }
1.  //save by B.java
2.  package mypack;
3.  import pack.*;
4.
5.  class B{
6.    public static void main(String args[]){
7.     A obj = new A();
8.     obj.msg();
9.    }
10. }
```

Output:Hello

2) Using packagename.classname

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

 Example of package by import package.classname

```
1.  //save by A.java
2.
3.  package pack;
4.  public class A{
5.    public void msg(){System.out.println("Hello");}
6.  }
1.  //save by B.java
2.  package mypack;
3.  import pack.A;
4.
5.  class B{
6.    public static void main(String args[]){
7.     A obj = new A();
8.     obj.msg();
9.    }
10. }
```

Output:Hello

3) Using fully qualified name

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

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

Example of package by import fully qualified name

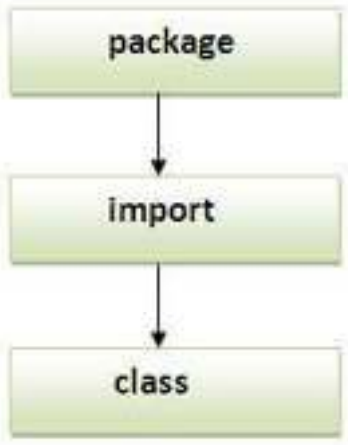
```
1.  //save by A.java
2.  package pack;
3.  public class A{
4.    public void msg(){System.out.println("Hello");}
5.  }
1.  //save by B.java
2.  package mypack;
3.  class B{
4.    public static void main(String args[]){
5.     pack.A obj = new pack.A();//using fully qualified name
6.     obj.msg();
7.    }
8.  }
```

Output:Hello

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



10.3 Subpackage in Java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

Example of Subpackage

```
1.  package com.javatpoint.core;
2.  class Simple{
3.    public static void main(String args[]){
4.     System.out.println("Hello subpackage");
5.    }
6.  }
```

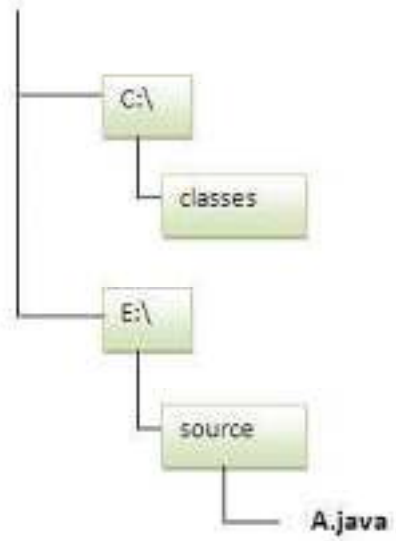
To Compile: javac -d . Simple.java

To Run: java com.javatpoint.core.Simple

Output:Hello subpackage

10.4 How to Send the Class File to Another Directory or Drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:




```
1.  //save as Simple.java
2.  package mypack;
3.  public class Simple{
4.   public static void main(String args[]){
5.      System.out.println("Welcome to package");
6.    }
7.  }
```

To Compile:

e:\sources> javac -d c:\classes Simple.java

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file reside

e:\sources> set classpath=c:\classes;.;

e:\sources> java mypack.Simple

Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

e:\sources> java -classpath c:\classes mypack.Simple

Output:Welcome to package

10.5 Ways to Load the Class Files or Jar Files

There are two ways to load the class files temporary and permanent.

- Temporary
  - By setting the classpath in the command prompt
  - By -classpath switch
- Permanent
  - By setting the classpath in the environment variables
  - By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

Rule: There can be only one public class in a java source file and it must be saved by the public class name.

```
1. //save as C.java otherwise Compilte Time Error
2.
3. class A{}
4. class B{}
5. public class C{}
```

How to put two public classes in a package?

If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:

```
1. //save as A.java
2.
3. package javatpoint;
4. public class A{}
1. //save as B.java
2.
3. package javatpoint;
4. public class B{}
```

One of the most innovative Packages in Java are a way to encapsulate a group of classes, interfaces, enumerations, annotations, and sub-packages. Conceptually, you can think of java packages as being similar to different folders on your computer. In this tutorial, we will cover the basics of packages in features of Java is the concept of packages. Java.

10.6 What is Package in Java?

Java package is a mechanism of grouping similar types of classes, interfaces, and sub-classes collectively based on functionality. When software is written in the Java programming language, it can be composed of hundreds or even thousands of individual classes. It makes sense to keep things organized by placing related classes and interfaces into packages.

Using packages while coding offers a lot of advantages like:

- **Re-usability:** The classes contained in the packages of another program can be easily reused
- help us to uniquely identify a class, for example, we can have **Name Conflicts:** Packages *company.sales.Employee* and *company.marketing.Employee* classes
- **Controlled Access:** Offers access protection such as protected classes, default classes, and private class
- **Data Encapsulation:** They provide a way to hide classes, preventing other programs from accessing classes that are meant for internal use only
- **Maintainance:** With packages, you can organize your project better and easily locate related classes

It's a good practice to use packages while coding in Java. As a programmer, you can easily figure out the classes, interfaces, enumerations, and annotations that are related. We have two types of packages in java.

Types of Packages in Java

Based on whether the package is defined by the user or not, packages are divided into two categories:

1. Built-in Packages
2. User-Defined Packages

Built-in Packages

Built-in packages or predefined packages are those that come along as a part of JDK (Java Development Kit) to simplify the task of Java programmer. They consist of a huge number of predefined classes and interfaces that are a part of Java API's. Some of the commonly used built-in packages are java.lang, java.io, java.util, java.applet, etc. Here's a simple program using a built-in

package. Built-in packages in Java provide a wide range of functionality for various tasks. Here's a brief overview of some commonly used built-in packages:

1. **java.lang**: This package is automatically imported in every Java program. It contains fundamental classes such as Object, String, Math, and wrappers for primitive types (Integer, Double, etc.).
2. **java.util**: This package contains utility classes and data structures like ArrayList, HashMap, HashSet, Date, Calendar, etc., used for various purposes such as collections, date, and time manipulation.
3. **java.io**: This package provides classes for input and output operations. It includes classes like FileInputStream, FileOutputStream, BufferedReader, BufferedWriter, etc., for reading from and writing to files and streams.
4. **java.net**: This package provides classes for networking operations. It includes classes like URL, HttpURLConnection, Socket, ServerSocket, etc., for creating network connections and communicating over the network.
5. **java.awt** and **javax.swing**: These packages provide classes for creating graphical user interfaces (GUIs). AWT (Abstract Window Toolkit) is the original GUI toolkit for Java, while Swing provides more advanced and customizable components.
6. **java.sql**: This package provides classes and interfaces for interacting with relational databases using SQL (Structured Query Language). It includes classes like Connection, Statement, ResultSet, etc., for database connectivity and querying.
7. **java.security**: This package provides classes and interfaces for implementing security-related functionality in Java applications. It includes classes for encryption, digital signatures, secure random number generation, and authentication.
8. **java.text**: This package provides classes for formatting and parsing textual data, such as numbers, dates, and messages. It includes classes like NumberFormat, DateFormat, MessageFormat, etc.

```
import javax.swing.*;

import javax.swing.table.DefaultTableModel;


public class TableExample {

   public static void main(String[] args) {

      // Data for the table

      Object[][] data = {

            {"John", 25, "Male"},

            {"Anna", 30, "Female"},

            {"Mark", 22, "Male"},

            {"Rachel", 28, "Female"}

      };


      // Column headers

      String[] columns = {"Name", "Age", "Gender"};


      // Create a default table model

      DefaultTableModel model = new DefaultTableModel(data, columns);


      // Create a JTable with the model
```

```
        JTable table = new JTable(model);

        // Create a JScrollPane to hold the table
        JScrollPane scrollPane = new JScrollPane(table);

        // Create a JFrame to hold the scroll pane
        JFrame frame = new JFrame("Table Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(scrollPane);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Sure, I can provide you with a basic example of how to create a table in Java using the Swing library. Here's a simple program that creates a JFrame with a JTable displaying some data:

java

Copy code

```
import javax.swing.*;
import javax.swing.table.DefaultTableModel;

public class TableExample {
    public static void main(String[] args) {
        // Data for the table
        Object[][] data = {
                {"John", 25, "Male"},
                {"Anna", 30, "Female"},
                {"Mark", 22, "Male"},
                {"Rachel", 28, "Female"}
        };

        // Column headers
        String[] columns = {"Name", "Age", "Gender"};

        // Create a default table model
        DefaultTableModel model = new DefaultTableModel(data, columns);

        // Create a JTable with the model
        JTable table = new JTable(model);

        // Create a JScrollPane to hold the table
        JScrollPane scrollPane = new JScrollPane(table);
```

```
        // Create a JFrame to hold the scroll pane
        JFrame frame = new JFrame("Table Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(scrollPane);
        frame.pack();
        frame.setVisible(true);
    }
}
```

This code creates a simple JFrame containing a JTable with four rows and three columns. The data for the table is provided as a 2D array, and the column headers are specified as an array of strings. The DefaultTableModel class is used to create a table model with the provided data and column headers. Finally, a JScrollPane is used to provide scrolling functionality for the table, and the table is added to the JFrame.

User-Defined Packages

User-defined packages are those which are developed by users in order to group related classes, interfaces, and sub-packages. With the help of an example program, let's see how to create packages, compile Java programs inside the packages, and execute them.

Creating a Package in Java

Creating a package in Java is a very easy task. Choose a name for the package and include a *package* command as the first statement in the Java source file. The java source file can contain the classes, interfaces, enumerations, and annotation types that you want to include in the package. For example, the following statement creates a package named ***MyPackage.***

```
package MyPackage;
```

The package statement simply specifies to which package the classes defined belongs to.

Note: *If you omit the package statement, the class names are put into the default package, which has no name. Though the default package is fine for short programs, it is inadequate for real applications.*

Including a Class in Java Package

To create a class inside a package, you should declare the package name as the first statement of your program. Then include the class as part of the package. But, remember that, a class can have only one package declaration. Here's a simple program to understand the concept.

```
package com.example;

import javax.swing.*;
import javax.swing.table.DefaultTableModel;


public class TableExample {
    public static void main(String[] args) {
        // Data for the table
        Object[][] data = {
                {"John", 25, "Male"},
                {"Anna", 30, "Female"},
                {"Mark", 22, "Male"},
                {"Rachel", 28, "Female"}
        };
```

```
        // Column headers
        String[] columns = {"Name", "Age", "Gender"};

        // Create a default table model
        DefaultTableModel model = new DefaultTableModel(data, columns);

        // Create a JTable with the model
        JTable table = new JTable(model);

        // Create a JScrollPane to hold the table
        JScrollPane scrollPane = new JScrollPane(table);

        // Create a JFrame to hold the scroll pane
        JFrame frame = new JFrame("Table Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(scrollPane);
        frame.pack();
        frame.setVisible(true);
    }
}
```

10.7 Creating a Class Inside a Package while Importing Another Package

Well, it's quite simple. You just need to import it. Once it is imported, you can access it by its name. Here's a sample program demonstrating the concept.

```
// File: com/util/UtilityClass.java
package com.util;

public class UtilityClass {
    public void doSomething() {
        System.out.println("Utility class is doing something.");
    }
}
// File: com/util/UtilityClass.java
package com.util;

public class UtilityClass {
    public void doSomething() {
        System.out.println("Utility class is doing something.");
    }
}
```

Ensure that you have the following directory structure:

project │ └──com │ ├──example │ MyClass.java │ └──util UtilityClass.java

To compile and run this code:

1. Open a terminal.
2. Navigate to the directory containing the **project** folder.
3. Compile the code using **javac com/example/MyClass.java com/util/UtilityClass.java**.
4. Run the compiled code using **java com.example.MyClass**.

This will create an instance of **UtilityClass** inside **MyClass** and call its **doSomething()** method.

Static Import in Java

Static import feature was introduced in Java from version 5. It facilitates the Java programmer to access any static member of a class directly without using the fully qualified name.

package MyPackage;**import static** java.lang.Math.*; //static

import**import static** java.lang.System.*;//

static import**public class** StaticImportDemo

{

public static void main(String args[]) {**double** val = 64.0;**double** sqroot = sqrt(val); // Access sqrt() method directlyout.println("Sq. root of " + val + " is " + sqroot);//We don't need to use 'System.out}}**Output:**Sq. root of 64.0 is 8.0

Though using static import involves less coding, overusing it might make the program unreadable and unmaintainable. Now let's move on to the next topic, access control in packages.

Access Protection in Java Packages

You might be aware of various aspects of Java's access control mechanism and its access specifiers. Packages in Java add another dimension to access control. Both classes and packages are a means of Data Encapsulation. While packages act as containers for classes and other subordinate packages, classes act as containers for data and code. Because of this interplay between packages and classes, Java packages addresses four categories of visibility for class members:

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

The table below gives a real picture of which type access is possible and which is not when using packages in Java:

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

We can simplify the data in the above table as follows:

1. Anything declared public can be accessed from anywhere
2. Anything declared private can be seen only within that class
3. If access specifier is not mentioned, an element is visible to subclasses as well as to other classes in the same package
4. Lastly, anything declared protected element can be seen outside your current package, but only to classes that subclass your class directly

This way, Java packages provide access control to the classes. Well, this wraps up the concept of packages in Java. Here are some points that you should keep in mind when using packages in Java.

Points to Remember

- Every class is part of some package. If you omit the package statement, the class names are put into the default package
- A class can have only one package statement but it can have more than one import package statements
- The name of the package must be the same as the directory under which the file is saved
- When importing another package, package declaration must be the first statement, followed by package import

Java packages are essential for organizing and managing classes and interfaces in large-scale software projects. They provide a hierarchical structure for grouping related components, enhancing code organization, reusability, and maintainability. In this document, we'll explore the creation and importation of packages in Java, adding classes to packages, and an introduction to the Java API.

10.8 Creating and Importing Packages

Creating a package in Java is a straightforward process. To create a package, you include a package statement at the beginning of your Java source file. This statement declares the package to which the class belongs. For example:

java

```
package com.example.myapp;
```

This statement indicates that the class belongs to the com.example.myapp package. It's important to follow the naming conventions for packages, which typically use reverse domain name notation to ensure uniqueness and avoid naming conflicts.

Once you've created a package, you can import it into other classes using the import statement. For example:

java

import com.example.myapp.MyClass;

This statement imports the MyClass class from the com.example.myapp package into the current class, allowing you to use MyClass without fully qualifying its name.

Adding Classes to Packages

To add a class to a package, simply include the package statement at the beginning of the Java source file, specifying the package name. For example:

java

Copy code

```
package com.example.myapp;

public class MyClass {
    // Class implementation
}
```

This code defines a class named MyClass within the com.example.myapp package. All classes in the same file with the same package statement belong to the specified package.

You can organize classes into sub-packages to further structure your codebase. For example:

java

Copy code

```
package com.example.myapp.util;

public class Helper {
    // Class implementation
}
```

Here, the Helper class is placed in the util sub-package of the com.example.myapp package.

Introduction to Java API

The Java API (Application Programming Interface) is a collection of pre-written classes, interfaces, and packages provided by Oracle Corporation. It offers a wide range of functionalities for developing Java applications, including file I/O, networking, GUI development, database connectivity, and more.

The Java API is organized into packages, such as java.lang, java.util, java.io, java.net, etc. These packages contain numerous classes and interfaces that developers can utilize to build robust and feature-rich applications.

For example, the java.util package provides classes like ArrayList, HashMap, LinkedList, etc., which are commonly used for data manipulation and storage. Similarly, the java.io package offers classes like File, InputStream, OutputStream, etc., for handling input and output operations.

To use classes from the Java API, you typically import the required packages into your Java source files using the import statement. For example:

java

Copy code

import java.util.ArrayList;

import java.util.HashMap;

This allows you to access the ArrayList and HashMap classes from the java.util package within your code.

In summary, packages play a crucial role in organizing Java code, facilitating code reuse, and enhancing modularity. By creating and importing packages, adding classes to packages, and leveraging the Java API, developers can efficiently build complex and scalable Java applications.

Summary

- In Java, packages provide a structured way to organize code by grouping related classes and interfaces together. When creating packages, developers define a directory structure that mirrors the package hierarchy, enabling easy navigation and management of code files.
- Through the use of the import statement, classes and interfaces from one package can be accessed and utilized in another.
- This facilitates code reusability, simplifies maintenance, and enhances the overall organization of large-scale Java projects. Additionally, packages foster modularity and encapsulation, promoting clean and maintainable codebases.
- The Java API, a cornerstone of Java development, comprises a comprehensive collection of classes and interfaces that cover a broad spectrum of functionalities.
- Ranging from basic utilities like data structures and I/O operations to advanced features like graphical user interfaces and networking, the Java API offers developers a rich set of tools to build diverse applications efficiently.
- Leveraging the Java API allows developers to expedite development cycles by tapping into pre-built, standardized components, reducing the need to implement common functionalities from scratch. With its extensive documentation and robustness, the Java API empowers developers to create scalable, reliable, and platform-independent software solutions across various domains and industries.

Keywords

Packages: In Java, packages are containers used to organize classes and interfaces into namespaces. They serve to logically group related code elements together, aiding in code organization, readability, and maintenance. Packages prevent naming conflicts and allow for modular development, facilitating code reuse across projects.

Import Statement: The import statement in Java is used to bring classes or entire packages into scope, allowing them to be referenced and utilized within a Java source file. It enables developers to access classes and interfaces defined in other packages without having to provide fully qualified names for each usage, thus improving code readability and reducing redundancy.

Java API (Application Programming Interface): The Java API is a collection of pre-written code and libraries provided by Java developers to facilitate software development. It comprises classes, interfaces, methods, and constants that cover a wide range of functionalities, including I/O operations, data manipulation, networking, and user interface creation. Developers leverage the Java API to build applications efficiently by utilizing existing, standardized components.

Classpath: The classpath in Java is an environment variable or command-line argument that specifies the location(s) where the Java runtime environment should look for classes and resources. It enables the Java Virtual Machine (JVM) to locate and load classes referenced by a Java program during runtime, including those from user-defined packages and external libraries.

JAR (Java ARchive): A JAR file in Java is a compressed archive file format that bundles multiple Java class files, associated metadata, and resources into a single file. It serves as a portable packaging format for Java applications and libraries, facilitating distribution and deployment. JAR files are commonly used for packaging and distributing Java libraries, applications, and applets.

Self Assessment

1. Which keyword is used to import a specific class from a package in Java?

A. import

B. include

C. require

D. load

2. What is the purpose of the java.lang package in Java?

A. Input/output operations

B. Graphical user interface

C. Networking

D. Fundamental classes

3. Which package provides classes for reading and writing data to files in Java?

A. java.util

B. java.io

C. java.net

D. java.nio

4. What is the role of the javac command in Java?

A. Executes Java bytecode

B. Compiles Java source code

C. Packages Java classes into JAR files

D. Executes Java applications

5. Which package provides classes for handling date and time in Java?

A. java.util

B. java.sql

C. java.time

D. java.text

6. What does the java.util.Scanner class provide in Java?

A. Graphics rendering

B. Database connectivity

C. User input parsing
D. Network socket handling

7. Which package provides classes and interfaces for database connectivity in Java?
A. java.sql
B. java.util
C. java.io
D. java.net

8. What does the java.awt package provide in Java?
A. Database connectivity
B. Networking
C. Graphical user interface components
D. Date and time manipulation

9. Which package provides classes and interfaces for handling security-related operations in Java?
A. java.util
B. java.security
C. java.net
D. java.awt

10. Which package provides classes for creating and manipulating threads in Java?
A. java.util
B. java.io
C. java.lang
D. java.net

11. What is the purpose of the java.net package in Java?
A. Graphics rendering
B. Database connectivity
C. Network communication
D. Date and time manipulation

12. Which package provides classes for formatting and parsing textual data in Java?
A. java.sql
B. java.text
C. java.util
D. java.io

13. What does the java.nio package provide in Java?
A. Networking
B. Input/output operations
C. Date and time manipulation
D. New I/O APIs

14. Which package provides classes and interfaces for mathematical operations in Java?

A. java.lang

B. java.math

C. java.util

D. java.text

15. What does the java.util.regex package provide in Java?

A. Database connectivity

B. Graphics rendering

C. Regular expression matching

D. Date and time manipulation

Answers for Self Assessment

1.	A	2.	D	3.	B	4.	B	5.	C
6.	C	7.	A	8.	C	9.	B	10.	C
11.	C	12.	B	13.	D	14.	B	15.	C

Review Questions

1. Explain the concept of inheritance in Java and provide an example demonstrating its use.
2. Describe the difference between == and .equals() methods in Java. Provide examples to illustrate their usage.
3. Discuss the significance of the static keyword in Java. How is it used, and what are its implications in terms of memory management and code execution?
4. Explain the purpose and usage of the try-catch-finally block in exception handling in Java. Provide an example demonstrating its usage in handling exceptions.
5. Discuss the concept of multithreading in Java. Explain how multithreading can be implemented using the Thread class and the Runnable interface, and describe some common use cases for multithreading in Java applications.

Further Readings

Balagurusamy E. Programming with Java 3e Primer. New Delhi: Tata McGraw Publishers.

Schildt H. Java A Beginner's Guide, 3rd ed. New York: Mc-Graw Hill.

Arnold, K., Gosling, J., & Holmes, D. (2005). *The Java programming language*. Addison Wesley Professional.

Haggar, P. (2000). *Practical Java: programming language guide*. Addison-Wesley Professional. Online

Web Links

https://www.scaler.com/topics/java/packages-in-java/

https://medium.com/edureka/packages-in-java-edureka-7afdd58f9f33

Unit 11: Exception Handling

CONTENTS

Objectives

After this unit you will be able to:

- Understand the introduction to exception.
- Identify the built-in exception.
- Identify user defined exception.
- Learn the different types of access modifiers.

Introduction

Exception handling is a fundamental aspect of Java programming, allowing developers to gracefully manage unexpected situations that may occur during program execution. In this chapter, we will delve into the intricacies of exception handling in Java, covering topics such as the concept of exceptions, built-in and user-defined exceptions, and techniques for catching and throwing exceptions effectively. Through detailed explanations, examples, and diagrams, we aim to provide a comprehensive understanding of exception handling principles in the Java programming language.

11.1 Introduction To Exception

In Java, Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates

an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

Consider the following example:

```
public class Example {
   public static void main(String[] args) {
      try {
         int result = divide(10, 0); // Attempting to divide by zero
         System.out.println("Result: " + result);
      } catch (ArithmeticException e) {
         System.out.println("Error: Division by zero");
      }
   }
   public static int divide(int dividend, int divisor) {
      return dividend / divisor; // Potential division by zero
   }
}
```

In this example, the divide method attempts to divide by zero, resulting in an Arithmetic Exception, which is caught and handled in the catch block of the main method.

Major reasons why an exception Occurs

- Invalid user input
- Device failure
- Physical limitations (out-of-disk memory)
- Code errors
- Opening an unavailable file

Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

11.2 Difference between Error and Exception

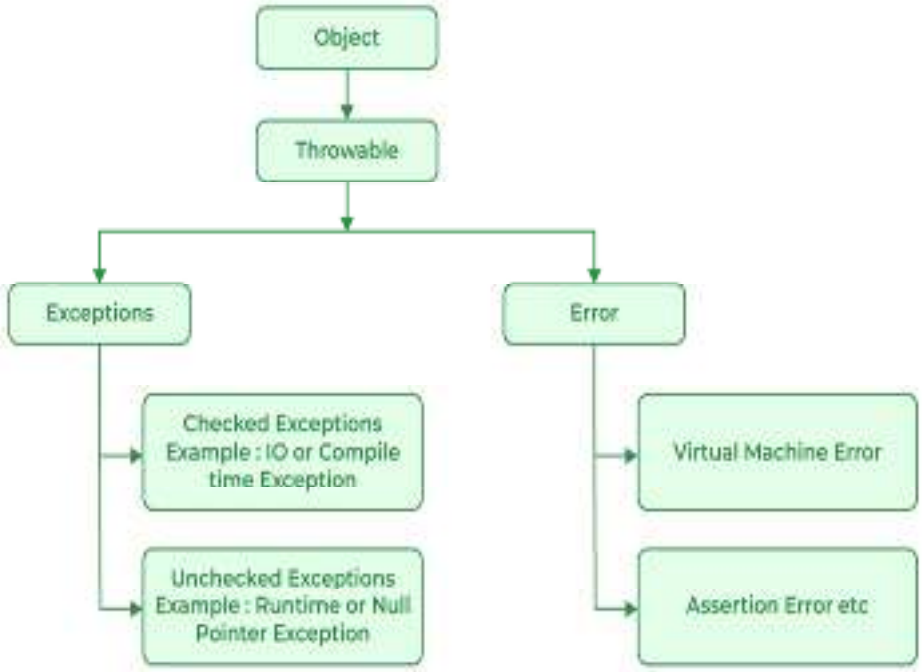
Let us discuss the most important part which is the differences between Error and Exception that is as follows:

Error: An Error indicates a serious problem that a reasonable application should not try to catch.

Exception: Exception indicates conditions that a reasonable application might try to catch.

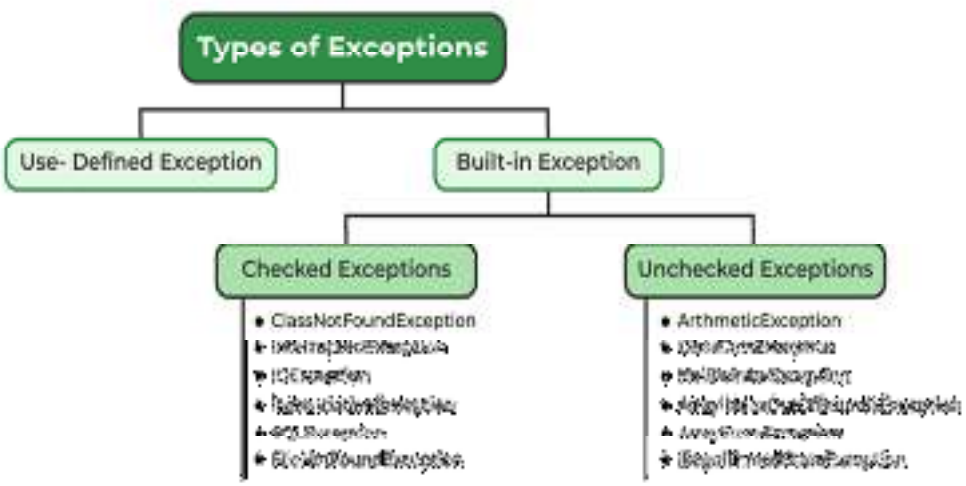
Exception Hierarchy

All exception and error types are subclasses of the class Throwable, which is the base class of the hierarchy. One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, Error is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.



11.3 Types of Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Exceptions can be categorized in two ways:

1. ***Built-in Exceptions***
   - ***Checked Exception***
   - ***Unchecked Exception***
2. ***User-Defined Exceptions***

Let us discuss the above-defined listed exception that is as follows:

1. Built-in Exceptions

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

Checked Exceptions: Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.

Unchecked Exceptions: The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

2. User-Defined Exceptions:

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called 'user-defined Exceptions'.

11.4 Built-in Exceptions

Java provides a wide range of built-in exceptions to handle common error scenarios. These exceptions are organized into a hierarchy, with the root class being Throwable. Common built-in exception classes include Arithmetic Exception, NullPointerException, and FileNotFoundException.

Let's illustrate this with an example:

```
import java.io.FileInputStream;

import java.io.FileNotFoundException;


public class FileExample {
   public static void main(String[] args) {
      try {
         FileInputStream file = new FileInputStream("nonexistent_file.txt");


// Attempting to open a non-existent file
      } catch (FileNotFoundException e) {
         System.out.println("Error: File not found");
      }
   }
}
```

Here, if the specified file does not exist, a FileNotFoundException is thrown

and caught in the catch block.

11.5 Catching Exceptions

Exception handling in Java revolves around the try-catch block, which allows developers to catch and handle exceptions that occur during program execution.

The try block contains the code that may potentially throw an exception, while the catch block provides a handler for the caught exception.

Consider the following example:

```
try {
   int result = divide(10, 0); // Attempting to divide by zero
```

```
    System.out.println("Result: " + result);
} catch (ArithmeticException e) {
    System.out.println("Error: Division by zero");
}
```

Here, the divide method attempts to divide by zero, resulting in

an ArithmeticException, which is caught and handled in the catch block.

11.6 Throwing Exceptions

In addition to catching exceptions, developers can also explicitly throw exceptions using the throw keyword. Throwing exceptions allows developers to signal error conditions programmatically and terminate program execution if necessary.

Let's illustrate this with an example:

```
public class Example {
    public static void main(String[] args) {
        try {
            checkAge(-5); // Throwing a custom exception for negative age
        } catch (IllegalArgumentException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }

    public static void checkAge(int age) {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative");
        }
    }
}
```

In this example, the checkAge method throws an IllegalArgumentException if the provided age is negative, which is caught and handled in the catch block.

11.7 How to Use the Try-catch Clause?

```
try {
    // block of code to monitor for errors
    // the code you think can raise an exception
} catch (ExceptionType1 exOb) {
    // exception handler for ExceptionType1
} catch (ExceptionType2 exOb) {
    // exception handler for ExceptionType2
}
// optional
finally { // block of code to be executed after try block ends
```

}

Certain key points need to be remembered that are as follows:

In a method, there can be more than one statement that might throw an exception, So put all these statements within their own try block and provide a separate exception handler within their own catch block for each of them.

If an exception occurs within the try block, that exception is handled by the exception handler associated with it. To associate the exception handler, we must put a catch block after it. There can be more than one exception handler. Each catch block is an exception handler that handles the exception to the type indicated by its argument. The argument, ExceptionType declares the type of exception that it can handle and must be the name of the class that inherits from the Throwable class.

For each try block, there can be zero or more catch blocks, but only one final block.

The finally block is optional. It always gets executed whether an exception occurred in try block or not. If an exception occurs, then it will be executed after try and catch blocks. And if an exception does not occur, then it will be executed after the try block. The finally block in Java is used to put important codes such as clean-up code e.g., closing the file or closing the connection.

If we write System.exit in the try block, then finally block will not be executed.

The advantages of Exception Handling in Java are as follows:

- Provision to Complete Program Execution
- Easy Identification of Program Code and Error-Handling Code
- Propagation of Errors
- Meaningful Error Reporting
- Identifying Error Types
- Methods to print the Exception information:

1. printStackTrace()

This method prints exception information in the format of the Name of the exception: description of the exception, stack trace.

 **Example:**

```
//program to print the exception information using printStackTrace() method
import java.io.*;
 class GFG {
   public static void main (String[] args) {
     int a=5;
     int b=0;
       try{
         System.out.println(a/b);
       }
     catch(ArithmeticException e){
       e.printStackTrace();
     }
   }
}
```

Output

java.lang.ArithmeticException: / by zero

at GFG.main(File.java:10)

2. toString()

The toString() method prints exception information in the format of the Name of the exception: description of the exception.

 **Example:**

```
//program to print the exception information using toString() method


import java.io.*;
 class GFG1 {
   public static void main (String[] args) {
     int a=5;
     int b=0;
       try{
         System.out.println(a/b);
       }
     catch(ArithmeticException e){
       System.out.println(e.toString());
     }
   }
}
```

Output

java.lang.ArithmeticException: / by zero

3. getMessage()

The getMessage() method prints only the description of the exception.

 Example:

```
//program to print the exception information using getMessage() method


import java.io.*;
class GFG1 {
   public static void main (String [] args) {
     int a=5;
     int b=0;
       try{
         System.out.println(a/b);
       }
```

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

Output

/ by zero

11.8 How Does JVM Handle an Exception?

Default Exception Handling: Whenever inside a method, if an exception has occurred, the method creates an Object known as an Exception Object and hands it off to the run-time system(JVM). The exception object contains the name and description of the exception and the current state of the program where the exception has occurred. Creating the Exception Object and handling it in the run-time system is called throwing an Exception. There might be a list of the methods that had been called to get to the method where an exception occurred. This ordered list of methods is called Call Stack. Now the following procedure will happen.

The run-time system searches the call stack to find the method that contains a block of code that can handle the occurred exception. The block of the code is called an Exception handler.

The run-time system starts searching from the method in which the exception occurred and proceeds through the call stack in the reverse order in which methods were called.

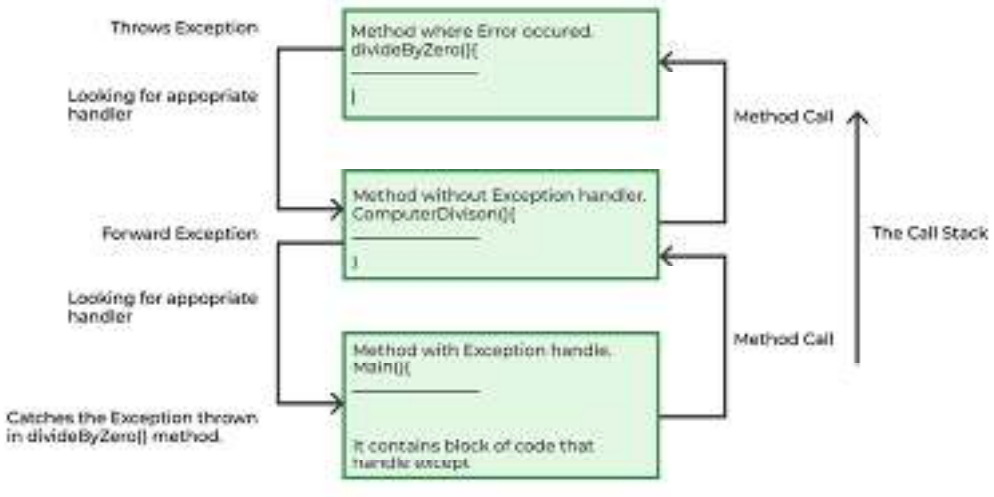
If it finds an appropriate handler, then it passes the occurred exception to it. An appropriate handler means the type of exception object thrown matches the type of exception object it can handle.

If the run-time system searches all the methods on the call stack and couldn't have found the appropriate handler, then the run-time system handover the Exception Object to the default exception handler, which is part of the run-time system. This handler prints the exception information in the following format and terminates the program abnormally.

Exception in thread "xxx" Name of Exception : Description

... . // Call Stack

Look at the below diagram to understand the flow of the call stack.



The Call Stack and searching the call stack for exception handler.

Illustration:

```
// Java Program to Demonstrate How Exception Is Thrown


// Class
// ThrowsExecp
class GFG {
   // Main driver method
   public static void main(String args[])
   {
      // Taking an empty string
      String str = null;
      // Getting length of a string
      System.out.println(str.length());
   }
}
```

Output

program output

Let us see an example that illustrates how a run-time system searches for appropriate exception handling code on the call stack.

 **Example:**

```
// Java Program to Demonstrate Exception is Thrown
// How the runTime System Searches Call-Stack
// to Find Appropriate Exception Handler


// Class
// ExceptionThrown
class GFG {


   // Method 1
   // It throws the Exception(ArithmeticException).
   // Appropriate Exception handler is not found
   // within this method.
   static int divideByZero(int a, int b)
   {
      // this statement will cause ArithmeticException
      // (/by zero)
      int i = a / b;
      return i;
   }
```

```
    // The runTime System searches the appropriate
  // Exception handler in method also but couldn't have
  // found. So looking forward on the call stack
  static int computeDivision(int a, int b)
  {
    int res = 0;
    // Try block to check for exceptions
    try {
        res = divideByZero(a, b);
    }
     // Catch block to handle NumberFormatException
    // exception Doesn't matches with
    // ArithmeticException
    catch (NumberFormatException ex) {
      // Display message when exception occurs
      System.out.println(
        "NumberFormatException is occurred");
    }
    return res;
  }
   // Method 2
  // Found appropriate Exception handler.
  // i.e. matching catch block.
  public static void main(String args[])
  {
     int a = 1;
    int b = 0;
     // Try block to check for exceptions
    try {
      int i = computeDivision(a, b);
    }
     // Catch block to handle ArithmeticException
    // exceptions
    catch (ArithmeticException ex) {
       // getMessage() will print description
      // of exception(here / by zero)
      System.out.println(ex.getMessage());
    }
  }
}
```

Output

/ by zero

11.9 How Programmer Handle an Exception?

Customized Exception Handling: Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Briefly, here is how they work. Program statements that you think can raise exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block.

Need for try-catch clause(Customized Exception Handling)

Consider the below program in order to get a better understanding of the try-catch clause.

 **Example:**

```
// Java Program to Demonstrate
// Need of try-catch Clause
// Class
class GFG {
   // Main driver method
   public static void main(String[] args)
   {
      // Taking an array of size 4
      int[] arr = new int[4];

      // Now this statement will cause an exception
      int i = arr[4];

      // This statement will never execute
      // as above we caught with an exception
      System.out.println("Hi, I want to execute");
   }
}
```

Output

program output

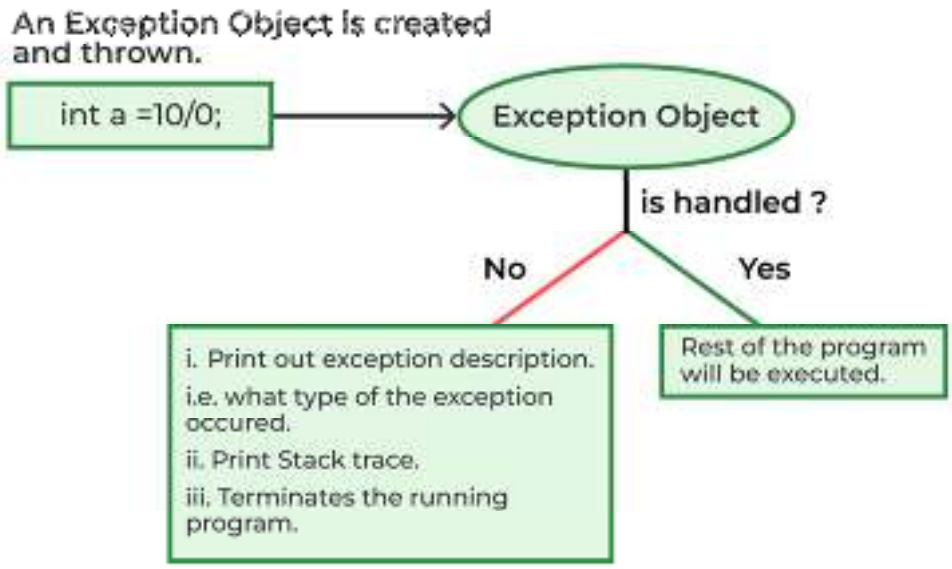
Output explanation: In the above example, an array is defined with size i.e. you can access elements only from index 0 to 3. But you trying to access the elements at index 4(by mistake) that's why it is throwing an exception. In this case, JVM terminates the program abnormally. The statement System.out.println("Hi, I want to execute"); will never execute. To execute it, we must handle the exception using try-catch. Hence to continue the normal flow of the program, we need a try-catch clause.

Summary

Exception handling is an essential aspect of Java programming, enabling developers to manage errors and ensure the reliability of their applications. By understanding the concepts of exceptions, leveraging built-in and user-defined exceptions, and mastering techniques for catching and throwing exceptions effectively, developers can write more robust and maintainable code. Effective exception handling is crucial for delivering high-quality software that meets user expectations.

Exception Handling in Java is one of the effective means to handle runtime errors so that the regular flow of the application can be preserved. Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

The summary is depicted via visual aid below as follows:



Keywords

Packages: In Java, packages are containers used to organize classes and interfaces into namespaces. They serve to logically group related code elements together, aiding in code organization, readability, and maintenance. Packages prevent naming conflicts and allow for modular development, facilitating code reuse across projects.

Import Statement: The import statement in Java is used to bring classes or entire packages into scope, allowing them to be referenced and utilized within a Java source file. It enables developers to access classes and interfaces defined in other packages without having to provide fully qualified names for each usage, thus improving code readability and reducing redundancy.

Java API (Application Programming Interface): The Java API is a collection of pre-written code and libraries provided by Java developers to facilitate software development. It comprises classes, interfaces, methods, and constants that cover a wide range of functionalities, including I/O operations, data manipulation, networking, and user interface creation. Developers leverage the Java API to build applications efficiently by utilizing existing, standardized components.

Classpath: The classpath in Java is an environment variable or command-line argument that specifies the location(s) where the Java runtime environment should look for classes and resources. It enables the Java Virtual Machine (JVM) to locate and load classes referenced by a Java program during runtime, including those from user-defined packages and external libraries.

JAR (Java ARchive): A JAR file in Java is a compressed archive file format that bundles multiple Java class files, associated metadata, and resources into a single file. It serves as a portable packaging format for Java applications and libraries, facilitating distribution and deployment. JAR files are commonly used for packaging and distributing Java libraries, applications, and applets.

Self Assessment

1. What is an exception in Java?

A. A syntax error

B. An unexpected event that occurs during the execution of a program

C. A logical error

D. A runtime error

2. Which keyword is used to handle exceptions in Java?

A. try

B. catch

C. throw

D. finally

3. What does the "finally" block do in exception handling?

A. Catches exceptions

B. Throws exceptions

C. Executes code whether an exception is thrown or not

D. Skips code execution

4. Which keyword is used to explicitly throw an exception in Java?

A. throw

B. catch

C. throws

D. try

5. What is the purpose of the "throws" keyword in Java?

A. To handle exceptions

B. To declare that a method may throw an exception

C. To catch exceptions

D. To create custom exceptions

6. Which keyword is used to rethrow an exception in Java?

A. rethrow

B. throw

C. catch

D. throws

7. Which of the following is NOT a type of exception in Java?

A. Checked exception

B. Unchecked exception

C. Runtime exception

D. Fatal exception

8. Which exception occurs when an inappropriate type is passed as an argument to a method?
a) NullPointerException
b) ClassCastException
c) IllegalArgumentException
d) ArrayIndexOutOfBoundsException

9. Which keyword is used to specify multiple exceptions in the throws clause?
A. multi
B. multiple
C. throw
D. comma-separated list of exception classes

10. What is the superclass of all exception types in Java?
A. Throwable
B. Exception
C. Error
D. RuntimeException

11. Which of the following is NOT a standard exception handling class in Java?
A. FileNotFoundException
B. NullPointerException
C. ExceptionThrown
D. ArrayIndexOutOfBoundsException

12. Which exception occurs when an array is accessed with an illegal index?
A. ArrayBoundsException
B. ArrayIndexOutOfBoundsException
C. ArrayIllegalIndexException
D. IndexOutOfBoundsException

13. What does the "throws" keyword do in a method signature?
A. Declares that the method will throw an exception
B. Catches exceptions within the method
C. Throws an exception explicitly
D. Specifies multiple catch blocks for the method

14. Which exception occurs when the Java Virtual Machine encounters an error it can't recover from?
A. ClassNotFoundException
B. StackOverflowError

C. OutOfMemoryError
D. VirtualMachineError

15. Which exception occurs when the compiler cannot find the class?
A. ClassNotFoundException
B. NoClassFoundException
C. ClassNotFound
D. ClassException

16. Which of the following is NOT a keyword related to exception handling in Java?
A. try
B. except
C. catch
D. finally

Answers for Self Assessment

1.	B	2.	A	3.	C	4.	A	5.	B
6.	B	7.	D	8.	C	9.	D	10.	A
11.	C	12.	B	13.	A	14.	D	15.	A
16.	B								

Review Questions

1. What is an exception in Java?
2. What is the purpose of exception handling in Java?
3. What are the two types of exceptions in Java? Differentiate between them.
4. Explain the difference between checked and unchecked exceptions in Java.
5. How do you handle exceptions in Java?
6. What happens if an exception is thrown inside a try block and there is no corresponding catch block to handle it?
7. What is the purpose of the finally block in exception handling? When does it execute?
8. Can you have multiple catch blocks following a single try block? If so, how?
9. Explain the difference between throw and throws keywords in Java exception handling.

Further Readings

Java: The Complete Reference, Eleventh Edition (PROGRAMMING & WEB DEV - OMG) Paperback – 19 March 201

Balagurusamy E. Programming with Java 3e Primer. New Delhi: Tata McGraw Publishers.

Schildt H. Java A Beginner's Guide, 3rd ed. New York: Mc-Graw Hill.

Arnold, K., Gosling, J., & Holmes, D. (2005). *The Java programming language*. Addison Wesley Professional.

Haggar, P. (2000). *Practical Java: programming language guide*. Addison-Wesley Professional. Online

Web Links

https://www.javatpoint.com/exception-handling-in-java

https://www.geeksforgeeks.org/exceptions-in-java/

Unit 12: More on Exception Handling

CONTENTS

Objectives

After this unit you will be able to:

- Understand the propagation of exceptions.
- Identify the difference between throws vs throw.
- Learn the handling of multiple exceptions.

Introduction

Before the publication of Java 7, we needed a unique catch block that could handle a specific exception. Due to this, there was a poor plan of action and extraneous blocks of code. A catch block is followed by either one or more catch blocks. Each catch block should contain different exception handlers. Java offers different types of catch blocks that handle different types of exceptions. A Multi catch block and a Single catch block are examples of such catch blocks.

A Java Multi-catch block is used primarily if one has to perform different tasks at the occurrence of various exceptions. One exception co-occurs, and only one catch block is executed. This catches block must have an order from the most specific to the most general. For example, the catch for an arithmetic exception should come before the catch for an exception.

12.1 Java Exception Propagation

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

 Example:

```
class TestExceptionPropagation1{
  void m(){
    int data=50/0;
```

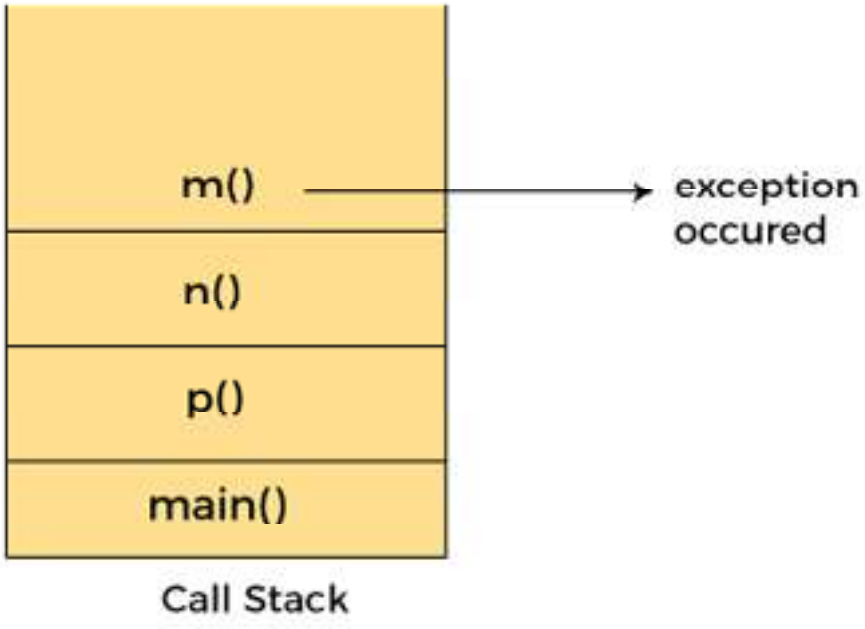
```
 }
 void n(){
  m();
 }
 void p(){
  try{
   n();
  }catch(Exception e){System.out.println("exception handled");}
 }
 public static void main(String args[]){
  TestExceptionPropagation1 obj=new TestExceptionPropagation1();
  obj.p();
  System.out.println("normal flow...");
 }
}
```

Output:

```
exception handled
    normal flow...
```

In the above example exception occurs in the m() method where it is not handled, so it is propagated to the previous n() method where it is not handled, again it is propagated to the p() method where exception is handled.

Exception can be handled in any method in call stack either in the main() method, p() method, n() method or m() method.



Exception Propagation Example

```
class TestExceptionPropagation2{
 void m(){
   throw new java.io.IOException("device error");//checked exception
```

```
 }
 void n(){
  m();
 }
 void p(){
  try{
   n();
  }catch(Exception e){System.out.println("exception handeled");}
 }
 public static void main(String args[]){
  TestExceptionPropagation2 obj=new TestExceptionPropagation2();
  obj.p();
  System.out.println("normal flow");
 }
}
```

Output:

Compile Time Error

12.2 Difference Between throw and throws in Java

The throw and throws are the concepts of exception handling in Java where the throw keyword throws the exception explicitly from a method or a block of code, whereas the throws keyword is used in the signature of the method.

Java throw

```
// Java program to demonstrate the working
// of throw keyword in exception handling


public class GFG {
        public static void main(String[] args)
        {
                // Use of unchecked Exception
                try {
                        // double x=3/0;
                        throw new ArithmeticException();
                }
                catch (ArithmeticException e) {
                        e.printStackTrace();
                }
}
```

Output:

```
java.lang.ArithmeticException
   at GFG.main(GFG.java:10)
```

Java throws

```
// Java program to demonstrate the working
// of throws keyword in exception handling
import java.io.*;
import java.util.*;

public class GFG {

	public static void writeToFile() throws Exception
	{
		BufferedWriter bw = new BufferedWriter(
			new FileWriter("myFile.txt"));
		bw.write("Test");
		bw.close();
	}

	public static void main(String[] args) throws Exception
	{
		try {
			writeToFile();
		}
		catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

Output:

```
java.security.AccessControlException: access denied ("java.io.FilePermission" "myFile.txt" "write")
  at GFG.writeToFile(GFG.java:10)
```

The differences between throw and throws in Java are:

S. No.	Key Difference	throw	throws
1.	Point of Usage	The **throw** keyword is used inside a function. It is used when it is required to throw an Exception logically.	The **throws** keyword is used in the function signature. It is used when the function has some statements that can lead to exceptions.
2.	Exceptions Thrown	The **throw** keyword is used to throw an exception explicitly. It can throw only one exception at a time.	The **throws** keyword can be used to declare multiple exceptions, separated by a comma. Whichever exception occurs, if matched with the declared ones, is thrown automatically then.

3.	Syntax	Syntax of **throw** keyword includes the instance of the Exception to be thrown. Syntax wise throw keyword is followed by the instance variable.	Syntax of **throws** keyword includes the class names of the Exceptions to be thrown. Syntax wise throws keyword is followed by exception class names.
4.	Propagation of Exceptions	**throw** keyword cannot propagate checked exceptions. It is only used to propagate the unchecked Exceptions that are not checked using the throws keyword.	**throws** keyword is used to propagate the checked Exceptions only.

12.3 Java Multi Catch Block

When using a multi-catch block, you first execute the try block. Once this is done, you will have two results, either an exception or no exception. In case you get an exception, go ahead and find an appropriate catch block to execute. In this situation, you can have various exceptions, for instance: executing the catch block for ExceptionType1, executing the catch block for ExceptionType2, executing the catch block for ExceptionTypen, and so on.

Once you complete executing the catch block for any ExceptionType, you execute the statement of a try-catch block. Once this is complete, the program comes to an end. If you executed the try block and got no exception, the statement out of the try-catch block is executed, and the program ends.

Let us take a piece of java code to have a better understanding:

<pre>public class MultipleCatchBlock {

```
   public static void main(String[] args) {
        try{
            int a[]=new int[7];
            a[7]=50/0;
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
   }
}
```

</pre>

```
<div class="wp-block-codemirror-blocks-code-block code-block">
<pre>public class MultipleCatchBlock {
   public static void main(String[] args) {
        try{
            int a[]=new int[7];
            a[7]=50/0;
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
   }
}</pre>
</div>
```

The output of the above code will be:

```
<div class="wp-block-codemirror-blocks-code-block code-block">
<pre>Arithmetic Exception occurs
rest of the code</pre>
</div>
```

In the code above, we have two exceptions for instance:

ArithmeticException, where we are trying to divide an integer by 0. That is, (50 / 0).

ArrayIndexOutOfBoundsException. This exception is because we are trying to assign a value to index 7 while we declare a new integer array with array bounds of 0 to 6, and we are also trying to assign a value to index 7. We use a duplicate code by publishing the message in both catch blocks.

The assignment operator, =, has a right-to-left associativity, so the ArithmeticException is first raised with the message divided by zero.

In the code above, the coding duplication has been decreased, whereas the efficiency has been increased since multiple exceptions have been caught in a single catch block.

Due to the lack of code repetition, the bytecode produced when the program was compiled will be smaller and superior to the bytecode produced when the program could have numerous catch blocks.

The catch parameter is implicitly final when a catch block manages several exceptions. This means we cannot give the catch parameters and values.

Java program to Catch the Base Exception

The rule is generalized to specialized when using a single catch block to catch multiple exceptions. This means that if there is a hierarchy of exceptions in the catch block, we can catch the base exception only instead of catching numerous specialized exceptions.

Let us take a look at the code shown below:

<pre>// a Java Program to catch the base exception class only

class Main {

 public static void main(String[] args) {

  try {

   int array[] = new int[7];

   array[7] = 50 / 0;

  } catch (Exception e) {

   System.out.println(e.getMessage());

  }

 }

}</pre>

The output of the code above is:

<pre>/ by zero</pre>

The exception classes in the code above are subclasses of the Exception class. Therefore, we can catch the Exception class instead of catching multiple specialised exceptions.

Let us see an example of handling the exception without maintaining the order of exceptions, from the most specific to the most general.

<pre>class MultipleCatchBlock{

 public static void main(String args[]){

  try{

   int a[]=new int[7];

   a[7]=50/0;

  }

  catch(Exception e){System.out.println("completed common task");}

  catch(ArithmeticException e){System.out.println("completed task 1");}

  catch(ArrayIndexOutOfBoundsException e){System.out.println("completed task 2");}

```
  System.out.println("rest of the code...");
 }
}</pre>
</div>
```

The output of the code is:

```
<div class="wp-block-codemirror-blocks-code-block code-block">
<pre>MultipleCatchBlock.java:8: error: exception ArithmeticException has already been caught
catch(ArithmeticException e){System.out.println("completed task 1");}
  ^
MultipleCatchBlock.java:9: error: exception ArrayIndexOutOfBoundsException has already been caught
  catch(ArrayIndexOutOfBoundsException e){System.out.println("completed task 2");}
  ^
2 errors
</pre>
</div>
```

Java program to catch the base and the child exception classes

In a case where the base exception class has already been specified in the catch block, you should not use the child exception classes in the same catch block. This is because it will result in a complication error.

Let us look at the code below:

```
<div class="wp-block-codemirror-blocks-code-block code-block">
<pre>// a java program to catch base and child exception class
class Main {
 public static void main(String[] args) {
  try {
   int array[] = new int[7];
   array[7] = 50 / 0;
  } catch (Exception | ArithmeticException | ArrayIndexOutOfBoundsException e) {
   System.out.println(e.getMessage());
  }
 }
}</pre>
</div>
```

The output of the code is:

```
<div class="wp-block-codemirror-blocks-code-block code-block">
<pre>/Main.java:6: error: Alternatives in a multi-catch statement cannot be related by subclassing
} catch (Exception | ArithmeticException | ArrayIndexOutOfBoundsException e) {
                   ^
 Alternative ArithmeticException is a subclass of alternative Exception
```

Main.java:6: error: Alternatives in a multi-catch statement cannot be related by subclassing

} catch (Exception | ArithmeticException | ArrayIndexOutOfBoundsException e) {

^

Alternative ArrayIndexOutOfBoundsException is a subclass of alternative Exception

</pre>

In the code above, we get a compilation error because both ArithmeticException and ArrayIndexOutOfBoundsException are both subclasses of the Exception class.

Let us take another example in Java

<pre>public class MultipleCatchBlock {

public static void main(String[] args) {

try{

int a[]=new int[5];

System.out.println(a[10]);

}

catch(ArithmeticException e)

{

System.out.println("Arithmetic Exception");

}

catch(ArrayIndexOutOfBoundsException e)

{

System.out.println("ArrayIndexOutOfBounds Exception");

}

catch(Exception e)

{

System.out.println("Parent Exception");

}

System.out.println("rest of the code");

}

} </pre>

The output of the code is:

<pre>ArrayIndexOutOfBounds Exception

rest of the code</pre>

There are two exceptions present in the try block in the code above. Only one exception occurs at a time; hence, the associated catch block is performed. The array size in the code is 5, but despite that, we are doing: System.out.println(a[10]); hence, we are getting ArrayIndexOutOfBounds Exception. Let us see what happens if we have multiple errors in the program:

<pre>public class MultipleCatchBlock {

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

      try{
          int a[]=new int[5];
          a[5]=30/0;
          System.out.println(a[10]);
         }
         catch(ArithmeticException e)
           {
            System.out.println("Arithmetic Exception");
           }
         catch(ArrayIndexOutOfBoundsException e)
           {
            System.out.println("ArrayIndexOutOfBounds Exception");
           }
         catch(Exception e)
           {
            System.out.println("Parent Exception");
           }
         System.out.println("rest of the code");
  }
} </pre>
```

The output of the code is:

<pre>Arithmetic Exception

rest of the code</pre>

In the program above, we are not getting the ArrayIndexOutOfBounds Exception even though we were printing a[10] when the array value was 5. This is because the arithmetic exception a[5]=30/0 occurs right before it; hence the catch block catches it and does not execute the code after it in the try block.

Summary

Exception propagation refers to movement of exception event from nested try or nested methods calls. A try block can be nested within another try block. Similarly a method can call another method where each method can handle exception independently or can throw checked/unchecked exceptions. Whenever an exception is raised within a nested try block/method, its exception is pushed to Stack. The exception propagates from child to parent try block or from child method to parent method and so on. If a method does not handle a checked exception, the method must declare it using the throws keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the throw keyword.

Multiple catch blocks in Java are used to catch/handle multiple exceptions that may be thrown from a particular code section. A try block can have multiple catch blocks to handle multiple exceptions.

Keywords

Packages: In Java, packages are containers used to organize classes and interfaces into namespaces. They serve to logically group related code elements together, aiding in code organization, readability, and maintenance. Packages prevent naming conflicts and allow for modular development, facilitating code reuse across projects.

Import Statement: The import statement in Java is used to bring classes or entire packages into scope, allowing them to be referenced and utilized within a Java source file. It enables developers to access classes and interfaces defined in other packages without having to provide fully qualified names for each usage, thus improving code readability and reducing redundancy.

Java API (Application Programming Interface): The Java API is a collection of pre-written code and libraries provided by Java developers to facilitate software development. It comprises classes, interfaces, methods, and constants that cover a wide range of functionalities, including I/O operations, data manipulation, networking, and user interface creation. Developers leverage the Java API to build applications efficiently by utilizing existing, standardized components.

Classpath: The classpath in Java is an environment variable or command-line argument that specifies the location(s) where the Java runtime environment should look for classes and resources. It enables the Java Virtual Machine (JVM) to locate and load classes referenced by a Java program during runtime, including those from user-defined packages and external libraries.

JAR (Java ARchive): A JAR file in Java is a compressed archive file format that bundles multiple Java class files, associated metadata, and resources into a single file. It serves as a portable packaging format for Java applications and libraries, facilitating distribution and deployment. JAR files are commonly used for packaging and distributing Java libraries, applications, and applets.

Self Assessment

1. When an exception is thrown in a method and not caught within that method, what happens?

A. The program terminates immediately.

B. The exception is propagated up the call stack until it is caught or the program terminates.

C. The exception is automatically caught by the Java runtime environment.

D. The exception is ignored and the program continues executing.

2. Which keyword is used to explicitly throw an exception in Java?

A. throw

B. throws

C. catch

D. try

3. What does the throws clause in a method signature indicate?

A. The method can throw multiple exceptions.

B. The method is expected to catch the specified exception.

C. The method propagates the specified exception to its caller.

D. The method handles the specified exception internally.

4. In Java, can a method declare that it throws more than one type of exception?

A. Yes

B. No

5. What happens if a method throws multiple exceptions of different types?

A. The caller must handle each exception type separately.

B. Only one of the exceptions is propagated to the caller.

C. The method must be modified to throw only one type of exception.

D. The compiler generates an error indicating ambiguity.

6. Which of the following is NOT a checked exception in Java?

A. IOException

B. NullPointerException

C. FileNotFoundException

D. ClassNotFoundException

7. When should you use a throws clause in a method signature?

A. When the method can potentially throw a checked exception.

B. When the method contains only checked exceptions.

C. When the method contains only unchecked exceptions.

D. When the method contains no exceptions.

8. Which statement is true about exception handling in Java?

A. All exceptions must be caught at compile-time.

B. Unchecked exceptions must be caught at runtime.

C. Checked exceptions must be caught or declared.

D. Errors are always caught by the try block.

9. In Java, can a method declare that it throws a checked exception without actually throwing it?

A. Yes

B. No

10. What happens if a method throws a checked exception but does not catch it?

A. The compiler generates an error.

B. The exception is automatically caught by the JVM.
C. The program compiles but may throw a runtime exception.
D. The method's caller must handle the exception.

11. Which statement is true about the order of catch blocks in a try-catch statement?
A. The catch blocks can be in any order.
B. The catch blocks must be ordered from the most specific to the most general exception type.
C. The catch blocks must be ordered from the most general to the most specific exception type.
D. The order of catch blocks does not matter as long as they cover all possible exceptions.

12. What does the finally block in a try-catch-finally statement do?
A. It catches exceptions thrown within the try block.
B. It executes regardless of whether an exception is thrown or caught.
C. It ensures that the program terminates immediately.
D. It rethrows any exceptions caught in the try block.

13. Which of the following statements is true about the throws keyword?
A. It is used to declare checked exceptions.
B. It is used to handle exceptions.
C. It is used to catch exceptions.
D. It is used to specify the exceptions that a method can catch.

14. Can the finally block be used without a catch block in Java?
A. Yes
B. No

15. Which statement is true about unchecked exceptions in Java?
A. They must be explicitly caught or declared.
B. They are subclasses of RuntimeException.
C. They are always caught at compile-time.
D. They are always caught at runtime.

16. Which exception is thrown when attempting to convert a String to a numeric type, but the String does not represent a valid number?
A. NumberFormatException
B. InvalidCastException
C. TypeConversionException
D. ConversionError

17. Which keyword is used to specify code that should be executed when an exception occurs?
A. try
B. catch
C. finally

D. throw

18. Can a method have both a throws clause and a try-catch block?

A. Yes

B. No

19. Which exception is thrown when trying to perform an illegal or inappropriate operation on an object?

A. UnsupportedOperationException

B. IllegalStateException

C. IllegalArgumentException

D. InvalidOperationException

20. In a try-catch-finally block, which block is optional?

A. try

B. catch

C. finally

D. Both catch and finally blocks are optional.

Self Assessment

1.	B	2.	A	3.	C	4.	A	5.	A
6.	B	7.	A	8.	C	9.	A	10.	A
11.	B	12.	B	13.	A	14.	A	15.	B
16.	A	17.	C	18.	A	19.	A	20.	C

Review Questions

1. What is exception propagation in Java?
2. Differentiate between checked and unchecked exceptions in Java.
3. Explain the purpose of the throws clause in Java method signatures.
4. When should you use the throw keyword in Java?
5. Describe the role of the try, catch, and finally blocks in Java exception handling.
6. Explain the difference between throw and throws in Java.
7. Is it possible for a method to have both a throws clause and a try-catch block? If yes, provide an example scenario.
8. Describe the order in which catch blocks are evaluated in a try-catch statement.
9. How are exceptions handled in a chain of method calls in Java?
10. Can you catch multiple exceptions in a single catch block in Java? If yes, provide an example.

11. Explain what happens if a checked exception is declared in a method's throws clause but is not actually thrown within the method.
12. When handling exceptions, what is the significance of the printStackTrace() method?

Further Readings

Java: The Complete Reference, Eleventh Edition (PROGRAMMING & WEB DEV - OMG) Paperback – 19 March 2019

Balagurusamy E. Programming with Java 3e Primer. New Delhi: Tata McGraw Publishers.

Schildt H. Java A Beginner's Guide, 3rd ed. New York: Mc-Graw Hill.

Arnold, K., Gosling, J., & Holmes, D. (2005). *The Java programming language*. Addison Wesley Professional.

Haggar, P. (2000). *Practical Java: programming language guide*. Addison-Wesley Professional. Online

Web Links

https://www.codeunderscored.com/java-catch-multiple-exceptions-explained-with-examples/

https://www.geeksforgeeks.org/difference-between-throw-and-throws-in-java/

https://www.javatpoint.com/exception-propagation

Unit 13: File Handling

CONTENTS

Objectives

- Understanding of Exception Handling:
- Proficiency in explaining the purpose and usage of the try-catch-finally block in exception handling.
- Ability to identify and handle exceptions effectively in Java

Introduction

File handling refers to the process of working with files stored on a computer's filesystem within a programming language. In Java, file handling involves tasks such as reading data from files, writing data to files, and performing various operations like creating, deleting, and modifying files and directories.

Java provides several built-in classes and interfaces in the java.io package to support file handling operations. These classes and interfaces offer functionalities for input and output operations, including reading from and writing to files, handling streams, and managing file-related exceptions.

13.1 File Handling

File handling is essential for many Java applications, especially those dealing with data storage, manipulation, and communication with external systems. It enables applications to interact with files stored locally or on remote servers, allowing for tasks such as data persistence, logging, configuration management, and data exchange between different systems. Overall, mastering file handling in Java is crucial for developing robust and efficient applications that require data storage and manipulation capabilities.

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

- **InPutStream** − The InputStream is used to read data from a source.
- **OutPutStream** − The OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one −

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file −

Example

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

public class CopyFile {

  public static void main(String args[]) throws IOException {
    FileInputStream in = null;
    FileOutputStream out = null;

    try {
      in = new FileInputStream("input.txt");
      out = new FileOutputStream("output.txt");

      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
      }
    }finally {
      if (in != null) {
        in.close();
      }
      if (out != null) {
```

```
         out.close();
       }
     }
   }
}
```

Now let's have a file **input.txt** with the following content −

This is test for copy file.

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

$javac CopyFile.java

$java CopyFile

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file −

 **Example**

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyFile {

  public static void main(String args[]) throws IOException {
    FileReader in = null;
    FileWriter out = null;

    try {
      in = new FileReader("input.txt");
      out = new FileWriter("output.txt");

      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
      }
    }finally {
      if (in != null) {
        in.close();
```

```
      }
      if (out != null) {
        out.close();
      }
    }
  }
}
```

Now let's have a file **input.txt** with the following content −

This is test for copy file.

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

$javac CopyFile.java

$java CopyFile

13.2 Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams −

- **Standard Input** − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
- **Standard Output** − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.
- **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

Following is a simple program, which creates **InputStreamReader** to read standard input stream until the user types a "q" −

 **Example**

```
import java.io.InputStreamReader;

public class ReadConsole {
  public static void main(String args[]) throws IOException {
    InputStreamReader cin = null;


    try {
      cin = new InputStreamReader(System.in);
      System.out.println("Enter characters, 'q' to quit.");
      char c;
      do {
        c = (char) cin.read();
        System.out.print(c);
```

```
      } while(c != 'q');
    }finally {
      if (cin != null) {
        cin.close();
      }
    }
  }
}
```

Let's keep the above code in ReadConsole.java file and try to compile and execute it as shown in the following program. This program continues to read and output the same character until we press 'q' −

$javac ReadConsole.java

$java ReadConsole

Enter characters, 'q' to quit.

1

1

e

e

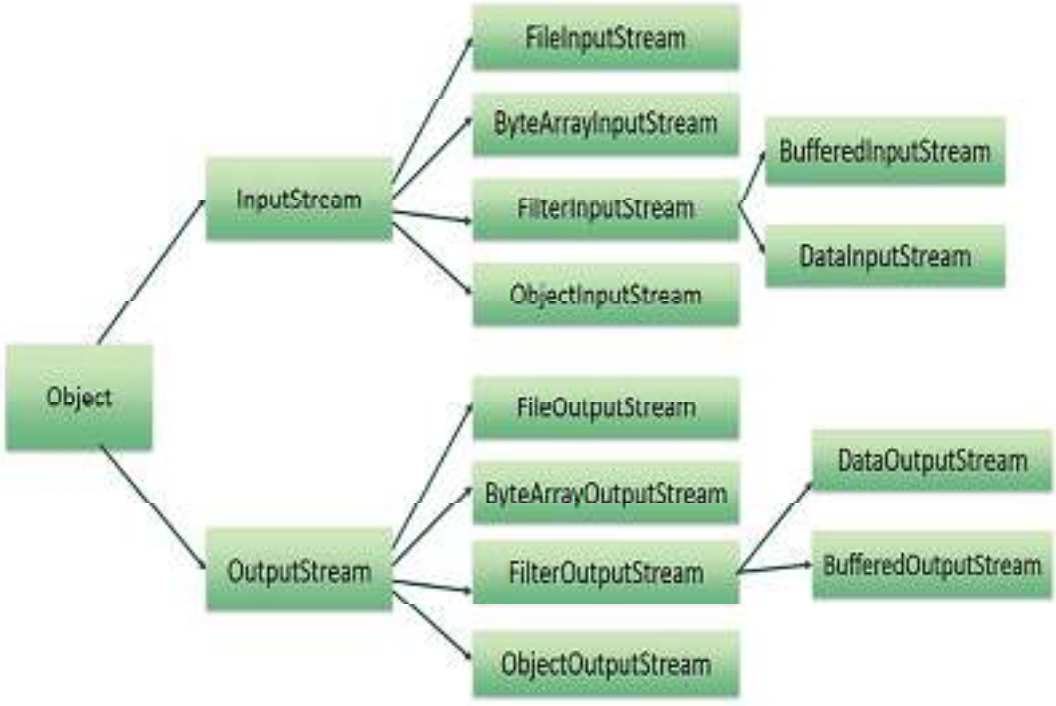
q

q

Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are **FileInputStream** and **FileOutputStream**, which would be discussed in this tutorial.

13.3 FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file −

InputStream f = new FileInputStream("C:/java/hello");

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows −

File f = new File("C:/java/hello");

InputStream f = new FileInputStream(f);

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

Sr.No.	Method & Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public int read(int r)throws IOException{}** This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's the end of the file.
4	**public int read(byte[] r) throws IOException{}** This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned.
5	**public int available() throws IOException{}** Gives the number of bytes that can be read from this file input stream. Returns an int.

There are other important input streams available, for more detail you can refer to the following links −

- ByteArrayInputStream
- DataInputStream

13.4 FileOutputStream

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file −

OutputStream f = new FileOutputStream("C:/java/hello")

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows −

File f = new File("C:/java/hello");

OutputStream f = new FileOutputStream(f);

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

Sr.No.	Method & Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public void write(int w)throws IOException{}** This methods writes the specified byte to the output stream.
4	**public void write(byte[] w)** Writes w.length bytes from the mentioned byte array to the OutputStream.

There are other important output streams available, for more detail you can refer to the following links −

- ByteArrayOutputStream
- DataOutputStream

Example

Following is the example to demonstrate InputStream and OutputStream −

```
import java.io.OutputStream;


public class fileStreamTest {


  public static void main(String args[]) {


    try {
      byte bWrite [] = {11,21,3,40,5};
      OutputStream os = new FileOutputStream("test.txt");
      for(int x = 0; x < bWrite.length ; x++) {
        os.write( bWrite[x] );   // writes the bytes
      }
```

```
      os.close();
          InputStream is = new FileInputStream("test.txt");
      int size = is.available();

      for(int i = 0; i < size; i++) {
        System.out.print((char)is.read() + "  ");
      }
      is.close();
    } catch (IOException e) {
      System.out.print("Exception");
    }
  }
}
```

The above code would create file test.txt and would write given numbers in binary format. Same would be the output on the stdout screen.

13.5 File Navigation and I/O

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- File Class
- FileReader Class
- FileWriter Class

Directories in Java

A directory is a File which can contain a list of other files and directories. You use **File** object to create directories, to list down files available in a directory. For complete detail, check a list of all the methods which you can call on File object and what are related to directories.

Creating Directories

There are two useful **File** utility methods, which can be used to create directories −

- The **mkdir( )** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
- The **mkdirs()** method creates both a directory and all the parents of the directory.

Following example creates "/tmp/user/java/bin" directory −

Example

```
import java.io.File;

public class CreateDir {

  public static void main(String args[]) {
    String dirname = "/tmp/user/java/bin";
```

```
    File d = new File(dirname);
        // Create directory now.
    d.mkdirs();
  }
}
```

Compile and execute the above code to create "/tmp/user/java/bin".

Note − Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.

Listing Directories

You can use **list( )** method provided by **File** object to list down all the files and directories available in a directory as follows −

 **Example**

```
import java.io.File;


public class ReadDir {


  public static void main(String[] args) {
    File file = null;
    String[] paths;


    try {
      // create new file object
      file = new File("/tmp");


      // array of files and directory
      paths = file.list();


      // for each name in the path array
      for(String path:paths) {
        // prints filename and directory name
        System.out.println(path);
      }
    } catch (Exception e) {
      // if any error occurs
      e.printStackTrace();
    }
  }
}
```

This will produce the following result based on the directories and files available in your **/tmp** directory −

Output

test1.txt

test2.txt

ReadDir.java

ReadDir.class

13.6 Java Console Class

The Java Console class is be used to get input from console. It provides methods to read texts and passwords.

If you read password using Console class, it will not be displayed to the user.

The java.io.Console class is attached with system console internally. The Console class is introduced since 1.5.

Let's see a simple example to read text from console.

```
1. String text=System.console().readLine();
2. System.out.println("Text is: "+text);
```

13.7 Java Console Class Declaration

Let's see the declaration for Java.io.Console class:

```
1. public final class Console extends Object implements Flushable
```

Java Console class methods

Method	Description
Reader reader()	It is used to retrieve the reader object associated with the console
String readLine()	It is used to read a single line of text from the console.
String readLine(String fmt, Object... args)	It provides a formatted prompt then reads the single line of text from the console.
char[] readPassword()	It is used to read password that is not being displayed on the console.
char[] readPassword(String fmt, Object... args)	It provides a formatted prompt then reads the password that is not being displayed on the console.
Console format(String fmt, Object... args)	It is used to write a formatted string to the console output stream.

Console printf(String format, Object... args)	It is used to write a string to the console output stream.
PrintWriter writer()	It is used to retrieve the PrintWriter object associated with the console.
void flush()	It is used to flushes the console.

How to get the object of Console

System class provides a static method console() that returns the singleton instance of Console class.

1. **public static** Console console(){}

Let's see the code to get the instance of Console class.

1. Console c=System.console();

Java Console Example

```
1. import java.io.Console;
2. class ReadStringTest{
3. public static void main(String args[]){
4. Console c=System.console();
5. System.out.println("Enter your name: ");
6. String n=c.readLine();
7. System.out.println("Welcome "+n);
8. }
9. }
```

Output

Enter your name: Neeraj Mathur

Welcome Neeraj Mathur

Java Console Example to read password

```
1. import java.io.Console;
2. class ReadPasswordTest{
3. public static void main(String args[]){
4. Console c=System.console();
5. System.out.println("Enter password: ");
6. char[] ch=c.readPassword();
7. String pass=String.valueOf(ch);//converting char array into string
8. System.out.println("Password is: "+pass);
9. }
10. }
```

Output

Enter password:

Password is: 123

Summary

- Introduction: File handling refers to the process of working with files stored on a computer's filesystem within a programming language.
- Java File Handling: In Java, file handling involves tasks such as reading data from files, writing data to files, and performing operations like creating, deleting, and modifying files and directories.
- java.io Package: Java provides several built-in classes and interfaces in the java.io package to support file handling operations.
- Functionalities: These classes and interfaces offer functionalities for input and output operations, including reading from and writing to files, handling streams, and managing file-related exceptions.
- Common Operations: File handling enables tasks such as data persistence, logging, configuration management, and data exchange between different systems.
- Importance: Mastering file handling in Java is crucial for developing robust and efficient applications that require data storage and manipulation capabilities.

Keywords

File Handling: File handling refers to the process of working with files stored on a computer's filesystem within a programming language. In Java, file handling involves tasks such as reading data from files, writing data to files, and performing operations like creating, deleting, and modifying files and directories.

java.io Package: The java.io package in Java provides classes and interfaces to support input and output operations, including file handling. It includes classes like File, FileInputStream, FileOutputStream, BufferedReader, BufferedWriter, and interfaces like Serializable and Closeable.

Reading from Files: Reading from files involves retrieving data stored in a file and loading it into a Java program. This process typically utilizes classes such as FileReader, BufferedReader, Scanner, or FileInputStream to read data from text files or binary files.

Writing to Files: Writing to files involves saving data generated by a Java program to a file on the filesystem. This process often utilizes classes such as FileWriter, BufferedWriter, PrintWriter, or FileOutputStream to write data to text files or binary files.

Exception Handling: Exception handling is crucial in file handling to handle potential errors or exceptions that may occur during file operations. This involves using try-catch blocks to catch exceptions thrown by file handling operations and handling them appropriately to prevent program crashes or data loss.

File Operations: File operations refer to common tasks performed on files, such as creating files and directories, deleting files, renaming files, checking file existence, and obtaining file metadata (such as file size, last modified timestamp, etc.).

File Streams: File streams are data streams used to read from or write to files. In Java, file streams are represented by classes such as FileInputStream, FileOutputStream, FileReader, and FileWriter, which provide low-level access to file data for reading or writing purposes.

Self Assessment

1. Which keyword is used to import a specific class from a package in Java?

A. import
B. include
C. require
D. load

2. What is the purpose of the java.lang package in Java?
A. Input/output operations
B. Graphical user interface
C. Networking
D. Fundamental classes

3. Which package provides classes for reading and writing data to files in Java?
A. java.util
B. java.io
C. java.net
D. java.nio

4. What is the role of the javac command in Java?
A. Executes Java bytecode
B. Compiles Java source code
C. Packages Java classes into JAR files
D. Executes Java applications

5. Which package provides classes for handling date and time in Java?
A. java.util
B. java.sql
C. java.time
D. java.text

6. What does the java.util.Scanner class provide in Java?
A. Graphics rendering
B. Database connectivity
C. User input parsing
D. Network socket handling

7. Which package provides classes and interfaces for database connectivity in Java?
A. java.sql
B. java.util
C. java.io
D. java.net

8. What does the java.awt package provide in Java?
A. Database connectivity
B. Networking
C. Graphical user interface components
D. Date and time manipulation

9. Which package provides classes and interfaces for handling security-related operations in Java?

A. java.util
B. java.security
C. java.net
D. java.awt

10. Which package provides classes for creating and manipulating threads in Java?

A. java.util
B. java.io
C. java.lang
D. java.net

11. What is the purpose of the java.net package in Java?

A. Graphics rendering
B. Database connectivity
C. Network communication
D. Date and time manipulation

12. Which package provides classes for formatting and parsing textual data in Java?

A. java.sql
B. java.text
C. java.util
D. java.io

13. What does the java.nio package provide in Java?

A. Networking
B. Input/output operations
C. Date and time manipulation
D. New I/O APIs

14. Which package provides classes and interfaces for mathematical operations in Java?

A. java.lang
B. java.math
C. java.util
D. java.text

15. What does the java.util.regex package provide in Java?

A. Database connectivity
B. Graphics rendering
C. Regular expression matching
D. Date and time manipulation

Answers for Self Assessment

1.	A	2.	D	3.	B	4.	B	5.	C
6.	C	7.	A	8.	C	9.	B	10.	C
11.	C	12.	B	13.	D	14.	B	15.	C

Review Questions

1. Explain the concept of inheritance in Java and provide an example demonstrating its use.
2. Describe the difference between == and .equals() methods in Java. Provide examples to illustrate their usage.
3. Discuss the significance of the static keyword in Java. How is it used, and what are its implications in terms of memory management and code execution?
4. Explain the purpose and usage of the try-catch-finally block in exception handling in Java. Provide an example demonstrating its usage in handling exceptions.
5. Discuss the concept of multithreading in Java. Explain how multithreading can be implemented using the Thread class and the Runnable interface, and describe some common use cases for multithreading in Java applications.

Further Readings

Balagurusamy E. Programming with Java 3e Primer. New Delhi: Tata McGraw Publishers.

Schildt H. Java A Beginner's Guide, 3rd ed. New York: Mc-Graw Hill.

Arnold, K., Gosling, J., & Holmes, D. (2005). *The Java programming language*. Addison Wesley Professional.

Haggar, P. (2000). *Practical Java: programming language guide*. Addison-Wesley Professional. Online

Web Links

http://www.uop.edu.pk/ocontents/lec-14.pdf

https://www.scaler.com/topics/java/java-io-streams/

Unit 14: More on File Handling

CONTENTS

Objectives

1. Mastery of file stream classes for efficient file manipulation.
2. Ability to distinguish between sequential and random access methods.
3. Proficiency in error handling techniques and understanding of various file types for effective file management.

Introduction

File handling in Java is a fundamental aspect of programming, enabling applications to interact with files stored on disk. Whether it's reading data from a text file, writing output to a log file, or managing directories, Java provides a robust set of classes and methods to handle file operations efficiently. The core of Java's file handling revolves around classes like File, FileInputStream, FileOutputStream, and their counterparts in the java.nio.file package. These classes empower developers to create, delete, read, and write files, navigate directories, and retrieve metadata such as file size and last modified time. Additionally, Java's exception handling mechanisms ensure graceful error recovery during file operations, promoting robust and reliable file management in applications.

Beyond basic file operations, Java's NIO (New I/O) package offers advanced functionalities for more complex file handling tasks. With features like channels, buffers, and asynchronous I/O operations, developers can achieve higher performance and scalability in file processing tasks. The java.nio.file package introduces classes like Path, Files, and FileSystem, providing a modern and versatile API for manipulating files and directories. Proper resource management, facilitated by constructs like try-with-resources, ensures that file resources are efficiently handled, preventing resource leaks and promoting cleaner code. In essence, file handling in Java equips developers with

the tools to manage file-based data effectively, facilitating a wide range of applications from simple file processing tasks to sophisticated data management systems.

Java FileWriter and FileReader classes are used to write and read data from text files (they are Character Stream classes). It is recommended **not** to use the FileInputStream and FileOutputStream classes if you have to read and write any textual information as these are Byte stream classes.

14.1 FileWriter

FileWriter is useful to create a file writing characters into it.

- This class inherits from the OutputStream class.
- The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an OutputStreamWriter on a FileOutputStream.
- FileWriter is meant for writing streams of characters. For writing streams of raw bytes, consider using a FileOutputStream.
- FileWriter creates the output file if it is not present already.

Constructors:

- **FileWriter(File file) –** Constructs a FileWriter object given a File object.
- **FileWriter (File file, boolean append) –** constructs a FileWriter object given a File object.
- **FileWriter (FileDescriptor fd) –** constructs a FileWriter object associated with a file descriptor.
- **FileWriter (String fileName) –** constructs a FileWriter object given a file name.
- **FileWriter (String fileName, Boolean append) –** Constructs a FileWriter object given a file name with a Boolean indicating whether or not to append the data written.

Methods:

- **public void write (int c) throws IOException –** Writes a single character.
- **public void write (char [] stir) throws IOException –** Writes an array of characters.
- **public void write(String str)throws IOException –** Writes a string.
- **public void write(String str, int off, int len)throws IOException –** Writes a portion of a string. Here off is offset from which to start writing characters and len is the number of characters to write.
- **public void flush() throws IOException** flushes the stream
- **public void close() throws IOException** flushes the stream first and then closes the writer.

Reading and writing take place character by character, which increases the number of I/O operations and affects the performance of the system.**BufferedWriter** can be used along with FileWriter to improve the speed of execution. The following program depicts how to create a text file using FileWriter

- Java

```
// Creating a text File using FileWriter

import java.io.FileWriter;

import java.io.IOException;

class CreateFile
```

```
{
   public static void main(String[] args) throws IOException
   {
      // Accept a string
      String str = "File Handling in Java using "+
            " FileWriter and FileReader";

      // attach a file to FileWriter
      FileWriter fw=new FileWriter("output.txt");

      // read character wise from string and write
      // into FileWriter
      for (int i = 0; i < str.length(); i++)
         fw.write(str.charAt(i));

      System.out.println("Writing successful");
      //close the file
      fw.close();
   }
}
```

14.2 FileReader

FileReader is useful to read data in the form of characters from a 'text' file.

- This class inherited from the InputStreamReader Class.
- The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate. To specify these values yourself, construct an InputStreamReader on a FileInputStream.
- FileReader is meant for reading streams of characters. For reading streams of raw bytes, consider using a FileInputStream.

Constructors:

- **FileReader(File file) –** Creates a FileReader , given the File to read from
- **FileReader(FileDescripter fd) –** Creates a new FileReader , given the FileDescripter to read from
- **FileReader(String fileName) –** Creates a new FileReader , given the name of the file to read from

Methods:

- **public int read () throws IOException –** Reads a single character. This method will block until a character is available, an I/O error occurs, or the end of the stream is reached.
- **public int read(char[] cbuff) throws IOException –** Reads characters into an array. This method will block until some input is available, an I/O error occurs, or the end of the stream is reached.

- **public abstract int read(char[] buff, int off, int len) throws IOException –**Reads characters into a portion of an array. This method will block until some input is available, an I/O error occurs, or the end of the stream is reached. Parameters: cbuf – Destination buffer off – Offset at which to start storing characters len – Maximum number of characters to read
- **public void close() throws IOException** closes the reader.
- **public long skip(long n) throws IOException –**Skips characters. This method will block until some characters are available, an I/O error occurs, or the end of the stream is reached.
  Parameters:
  n – The number of characters to skip

The following program depicts how to read from the 'text' file using FileReader

- Java

```
// Reading data from a file using FileReader
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
class ReadFile
{
   public static void main(String[] args) throws IOException
   {
      // variable declaration
      int ch;

      // check if File exists or not
      FileReader fr=null;
      try
      {
         fr = new FileReader("text");
      }
      catch (FileNotFoundException fe)
      {
         System.out.println("File not found");
      }

      // read from FileReader till the end of file
      while ((ch=fr.read())!=-1)
         System.out.print((char)ch);

      // close the file
      fr.close();
```

```
    }
}
```

Check if a File is Hidden in Java

isHidden() method of File class in Java can use be used to check if a file is hidden or not. This method returns a boolean value – true or false.

Syntax:

public static boolean isHidden(Path path)

throws IOException

Parameters: Path to the file to test.

Return Type: A boolean value, true if file is found hidden else returns false as a file is not found hidden

Exceptions Thrown:

- IOException: If an I/O error occurs
- **SecurityException**: In the case of the default provider, and a security manager is installed, the checkRead() method is invoked to check read access to the file.

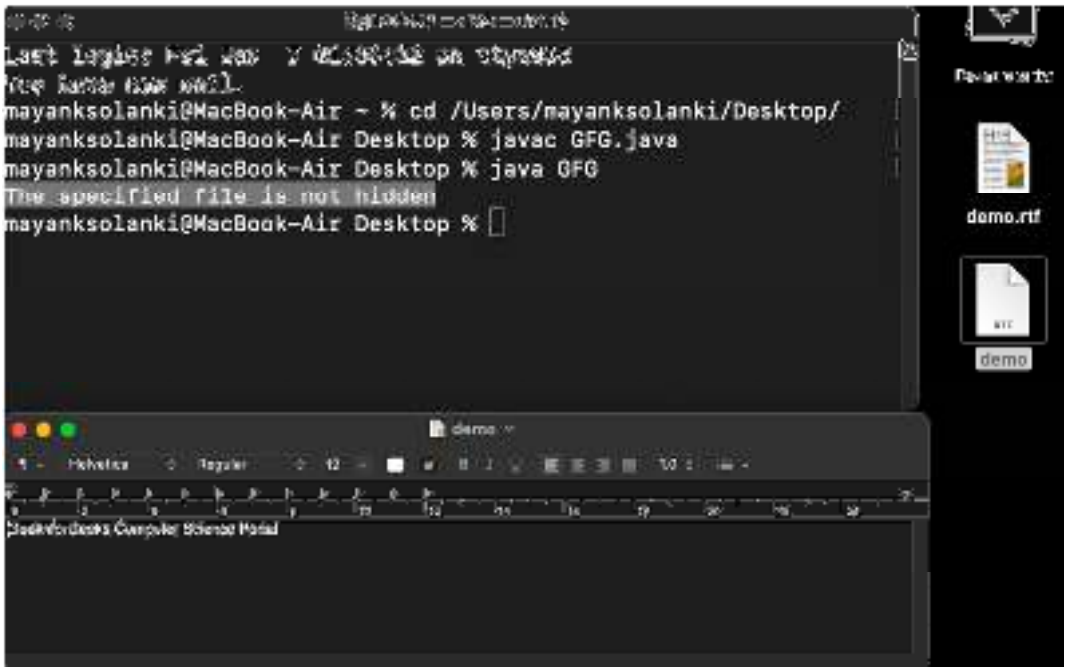
Remember: Depending on the implementation the isHidden() method may require to access the file system to determine if the file is considered hidden.

 **Example:**

- Java

```
// Java Program to Check if Given File is Hidden or Not
// Using isHidden() Method of File class

// Importing required classes
import java.io.File;
import java.io.IOException;

// Main class
// HiddenFileCheck
public class GFG {

  // Main driver method
  public static void main(String[] args)
    throws IOException, SecurityException
  {

    // Creating a file by
    // creating an object of File class
    File file = new File(
      "/users/mayanksolanki/Desktop/demo.rtf");

    // Checking whether file is hidden or not
    // using isHidden() method
    if (file.isHidden())

      // Print statement as file is found hidden
      System.out.println(
        "The specified file is hidden");
    else

      // Print statement as file is found as not
      // hidden
      System.out.println(
        "The specified file is not hidden");
  }
}
```

Output:



Output Explanation: As it can easily be visualized from the background of the output that the 'demo.rtf' file popping icon is clearly seen. The code reflects that a specific file is not hidden on the terminal output as seen above.

Note: The precise definition of hidden is a platform or provider-dependent.

- ***UNIX:*** *A file is hidden if its name begins with a period character ('.').*
- ***Windows:*** *A file is hidden if it is not a directory and the DOS hidden attribute is set.*

File Permissions in Java

Last Updated : 22 Apr, 2022

-
-
-

Java provides a number of method calls to check and change the permission of a file, such as a read-only file can be changed to have permissions to write. File permissions are required to be changed when the user wants to restrict the operations permissible on a file. For example, file permission can be changed from write to read-only because the user no longer wants to edit the file.

Checking the Current File Permissions

A file can be in any combination of the following permissible permissions depicted by methods below in tabular format/

Method	**Action Performed**
canExecutable()	Returns true if and only if the abstract pathname exists and the application is allowed to execute the file
canRead()	Tests whether the application can read the file denoted by this abstract pathname

Method	Action Performed
canWrite()	Returns true if and only if the file system actually contains a file denoted by this abstract pathname and the application is allowed to write to the file; false otherwise

Implementation: A file can be readable and writable but not executable. Here's a Java program to get the current permissions associated with a file.

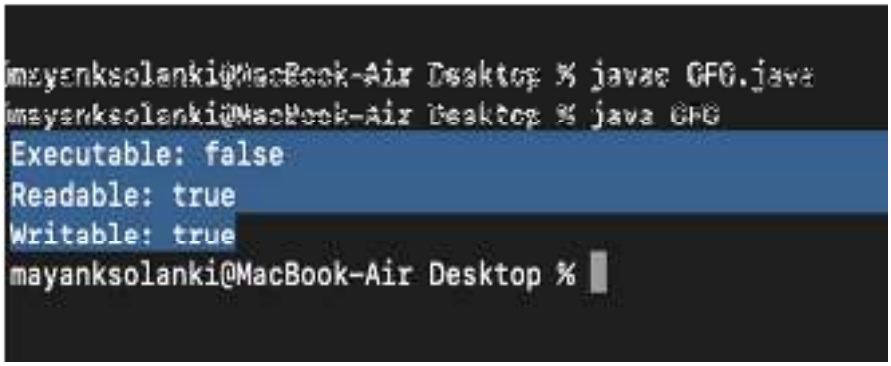
 **Example:**

- Java

```
// Java Program to Check the Current File Permissions

// Importing required classes
import java.io.*;

// Main class
public class Test {

  // Main driver method
  public static void main(String[] args)
  {

    // Creating a file by
    // creating object of File class
    File file
      = new File("C:\\Users\\Mayank\\Desktop\\1.txt");

    // Checking if the file exists
    // using exists() method of File class
    boolean exists = file.exists();
    if (exists == true) {

      // Printing the permissions associated
      // with the file
      System.out.println("Executable: "
                         + file.canExecute());
      System.out.println("Readable: "
                         + file.canRead());
```

```
            System.out.println("Writable: "
                               + file.canWrite());
        }

        // If we enter else it means
        // file does not exists
        else {
            System.out.println("File not found.");
        }
    }
}
```

Output:



14.3 Changing File Permissions

A file in Java can have any combination of the following permissions:

- Executable
- Readable
- Writable

Here are methods to change the permissions associated with a file as depicted in a tabular format below as follows:

Method	**Action Performed**
setExecutable()	Set the owner's execute permission for this abstract pathname
setReadable()	Set the owner's read permission for this abstract pathname
setWritable()	Set the owner's write permission for this abstract pathname

Note:

- setReadable() Operation will fail if the user does not have permission to change the access permissions of this abstract path name. If readable is false and the underlying file system does not implement a read permission, then the operation will fail.
- setWritable() Operation will fail if the user does not have permission to change the access permissions of this abstract pathname.

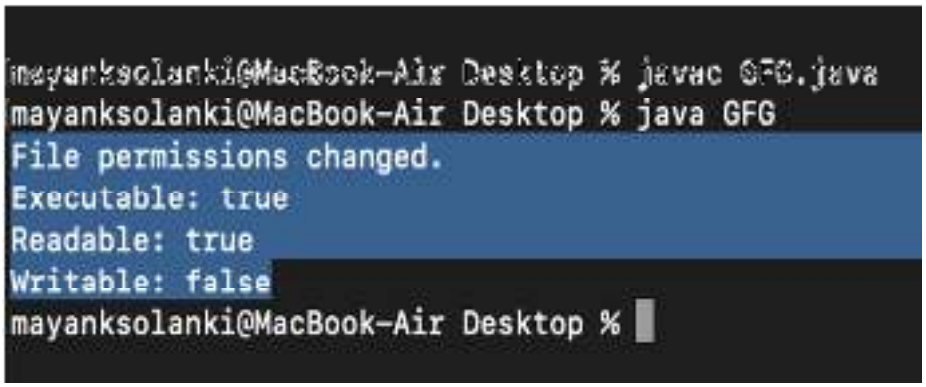
Example:

- Java

```
// Java Program to Change File Permissions

// Importing required classes
import java.io.*;

// Main class
public class GFG {

  // Main driver method
  public static void main(String[] args)
  {
    // Creating a new file by
    // creating object of File class where
    // local directory is passed as in argument
    File file
      = new File("C:\\Users\\Mayank\\Desktop\\1.txt");

    // Checking if file exists
    boolean exists = file.exists();
    if (exists == true) {

      // Changing the file permissions
      file.setExecutable(true);
      file.setReadable(true);
      file.setWritable(false);
      System.out.println("File permissions changed.");

      // Printing the permissions associated with the
      // file currently
      System.out.println("Executable: "
```

```
                + file.canExecute());
        System.out.println("Readable: "
                + file.canRead());
        System.out.println("Writable: "
                + file.canWrite());
    }

    // If we reach here, file is not found
    else {
        System.out.println("File not found");
    }
  }
}
```

Output:



14.4 FileWriter Class in Java

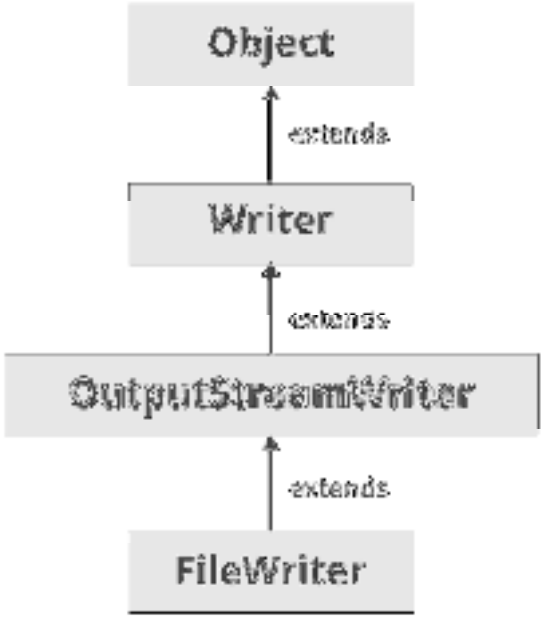
Last Updated : 10 Feb, 2022

-
-
-

Java FileWriter class of java.io package is used to write **data in character** form to file. Java FileWriter class is used to write character-oriented data to a file. It is a character-oriented class that is used for file handling in java.

- This class inherits from OutputStreamWriter class which in turn inherits from the Writer class.
- The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an OutputStreamWriter on a FileOutputStream.
- FileWriter is meant for writing streams of characters. For writing streams of raw bytes, consider using a FileOutputStream.
- FileWriter creates the output file if it is not present already.

14.5 Hierarchy of Java FileWriter Class



FileWriter extends OutputStreamWriter and Writer classes. It implements Closeable, Flushable, Appendable, AutoCloseable interfaces.

14.6 Java FileWriter Class Declaration

public class FileWriter extends OutputStreamWriter

Constructors of FileWriter Class

1. FileWriter(File file): It constructs a FileWriter object given a File object. It throws an **IOException** if the file exists but is a directory rather than a regular file does not exist but cannot be created, or cannot be opened for any other reason.

FileWriter fw = new FileWriter(File file);

2. FileWriter(File file, boolean append): It constructs a FileWriter object given a File object. If the second argument is true, then bytes will be written to the end of the file rather than the beginning. It throws an **IOException** if the file exists but is a directory rather than a regular file or does not exist but cannot be created, or cannot be opened for any other reason.

FileWriter fw = new FileWriter(File file, boolean append);

3. FileWriter(FileDescriptor fd): It constructs a FileWriter object associated with a file descriptor.

FileWriter fw = new FileWriter(FileDescriptor fd);

4. FileWriter(File file, Charset charset): It constructs the fileWriter when file and charset is given.

FileWriter fw = new FileWriter(File file, Charset charset);

5. FileWriter(File file, Charset charset, boolean append): It constructs the fileWriter when file and charset is given and a boolean indicating whether to append the data written or not.

FileWriter fw = new FileWriter(File file, Charset charset, boolean append);

6. FileWriter(String fileName): It constructs a FileWriter object given a file name.

FileWriter fw = new FileWriter(String fileName);

7. FileWriter(String fileName, Boolean append): It constructs a FileWriter object given a file name with a Boolean indicating whether or not to append the data written.

FileWriter fw = new FileWriter(String fileName, Boolean append);

8. FileWriter(String fileName, Charset charset): It constructs a FileWriter when a fileName and charset is given.

FileWriter fw = new FileWriter(String fileName, Charset charset);

9. FileWriter(String fileName, Charset charset, boolean append): It constructs a fileWriter when a fileName and a charset are given and a boolean variable indicating whether to append data or not.

FileWriter fw = new FileWriter(String fileName, Charset charset, boolean append);

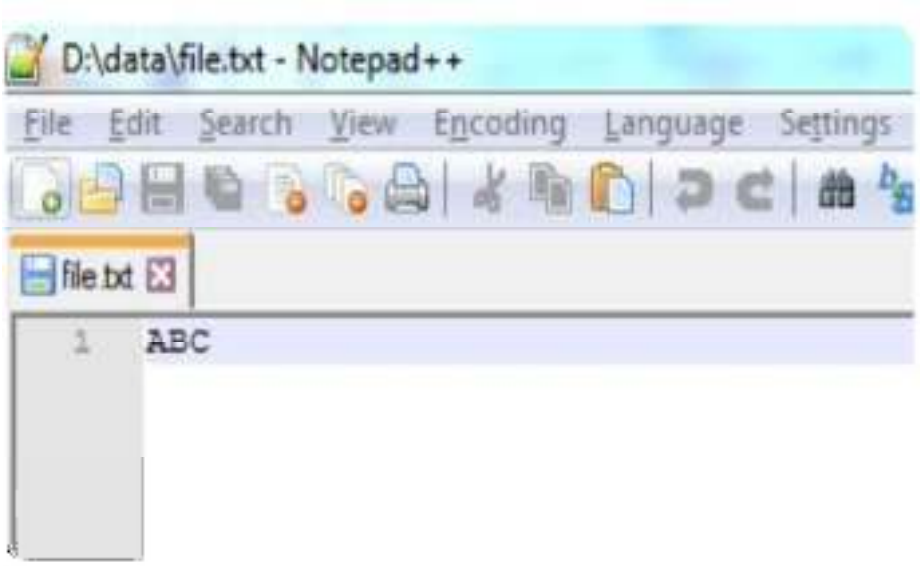
 **Example:**

- Java

```
// Java program to create a text File using FileWriter

import java.io.FileWriter;
import java.io.IOException;
import java.util.*;
class GFG {
   public static void main(String[] args)
      throws IOException
   {
      // initialize a string
      String str = "ABC";
      try {

         // attach a file to FileWriter
         FileWriter fw
            = new FileWriter("D:/data/file.txt");

         // read each character from string and write
         // into FileWriter
         for (int i = 0; i < str.length(); i++)
            fw.write(str.charAt(i));

         System.out.println("Successfully written");

         // close the file
         fw.close();
      }
      catch (Exception e) {
         e.getStackTrace();
      }
```

```
    }
}
```

Output:



14.7 Overwriting vs Appending the File

While creating a Java FileWriter, we can decide whether we want to append the file to an existing file, or we want to overwrite any existing file. This can be decided by choosing the appropriate constructor. The constructor for **overwriting** any existing file takes only **one parameter that is a file name.**

Writer fileWriter = new FileWriter("c:\\data\\output.txt");

The constructor for **appending the file** or overwriting the file, takes **two parameters, the file name and a boolean variable** which decides whether to append or overwrite the file

Writer fileWriter = new FileWriter("c:\\data\\output.txt", true); // appends to file

Writer fileWriter = new FileWriter("c:\\data\\output.txt", false); // overwrites file

Basic Methods of FileWriter Class

1. Write()

- **write(int a):** This method writes a single character specified by int a.
- **write(String str, int pos, int length):** This method writes a portion of the string from position **pos** until the **length** number of characters.
- **write(char ch[], int pos, int length):** This method writes the position of characters from array ch[] from position **pos** till **length** number of characters.
- **write(char ch[]):** This method writes an array of characters specified by ch[].
- **write(String st):** This method writes a string value specified by 'st' into the file.
- Java

```
// Java program to write text to file
```

```
import java.io.FileWriter;

public class GFG {

   public static void main(String args[])
   {

      String data = "Welcome to gfg";

      try {
         // Creates a FileWriter
         FileWriter output
            = new FileWriter("output.txt");

         // Writes the string to the file
         output.write(data);

         // Closes the writer
         output.close();
      }

      catch (Exception e) {
         e.getStackTrace();
      }
   }
}
```

Output:

The file output.txt will contain "**Welcome to gfg**" text.

2. getEncoding()

This method is used to get the type of encoding that is used for writing the data.

- Java

```
// java program to show the usage
// of getEncoding() function

import java.io.FileWriter;
import java.nio.charset.Charset;
```

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

      String file = "output.txt";

      try {
         // Creates a FileReader with default encoding
         FileWriter o1 = new FileWriter(file);

         // Creates a FileReader specifying the encoding
         FileWriter o2 = new FileWriter(
            file, Charset.forName("UTF11"));

         // Returns the character encoding of the reader
         System.out.println("Character encoding of o1: "
                        + o1.getEncoding());
         System.out.println("Character encoding of o2: "
                        + o2.getEncoding());

         // Closes the reader
         o1.close();
         o2.close();
      }

      catch (Exception e) {
         e.getStackTrace();
      }
   }
}
```

Output:

The character encoding of output1: Cp1253

The character encoding of output2: UTF11

In the above example, we have created 2 file writer named output1 and output2.

- **output1:** does not specify the character encoding. Hence, the getEncoding() method returns the default character encoding.
- **output2:** specifies the character encoding, **UTF11**. Hence, the getEncoding() method returns the specified character encoding.

3. close() method:

After finishing writing characters to a FileWriter, we should close it. And this is done by calling the close() method.

```
try {
   // Creates a FileReader with default encoding
   FileWriter o1 = new FileWriter(file);

   // Creates a FileReader specifying the encoding
   FileWriter o2 = new FileWriter(file, Charset.forName("UTF11"));

   // Returns the character encoding of the reader
   System.out.println("Character encoding of o1: " + o1.getEncoding());
   System.out.println("Character encoding of o2: " + o2.getEncoding());

   // Closes the FileWriter
   o1.close();
   o2.close();
}
```

14.8 FileWriter vs FileOutputStream

- FileWriter writes streams of characters while FileOutputStream is meant for writing streams of raw bytes.
- FileWriter deals with the character of 16 bits while on the other hand, FileOutputStream deals with 8-bit bytes.
- FileWriter handles Unicode strings while FileOutputStream writes bytes to a file and it does not accept characters or strings and therefore for accepting strings, it needs to be wrapped up with OutputStreamWriter.

Methods of FileWriter Class

S. No.	Method	Description
1.	void write(String text)	It is used to write the string into FileWriter.
2.	void write(char c)	It is used to write the char into FileWriter.
3.	void write(char[] c)	It is used to write a char array into FileWriter.

S. No.	Method	Description
4.	void flush()	It is used to flushes the data of FileWriter.
5.	void close()	It is used to close the FileWriter.

Methods of OutputStreamWriter Class

S. No.	Method	Description
1.	flush()	Flushes the stream.
2.	getEncoding()	Returns the name of the character encoding being used by this stream.
3.	write(char[] cbuf, int off, int len)	Writes a portion of an array of characters.
4.	write(int c)	Writes a single character.
5.	write(String str, int off, int len)	Writes a portion of a string.

Methods of Writer Class

S. No.	Method	Description
1.	append(char c)	Appends the specified character to this writer.

S. No.	Method	Description
2.	append(CharSequence csq)	Appends the specified character sequence to this writer.
3.	append(CharSequence csq, int start, int end)	Appends a subsequence of the specified character sequence to this writer.
4.	close()	Closes the stream, flushing it first.
5.	nullWriter()	Returns a new Writer which discards all characters.
6.	write(char[] cbuf)	Writes an array of characters.
7.	write(String str)	Writes a string.

Summary

- File handling in programming involves various techniques for interacting with files stored on a computer's storage system. One common approach is using file stream classes, which provide a set of functionalities for reading from and writing to files.
- These classes, such as ifstream and ofstream in C++, allow programmers to open files, perform operations like reading or writing data, and close files once done. By using file stream classes, developers can efficiently manipulate file contents, enabling tasks such as data processing, configuration management, and logging within their applications.
- Moreover, file handling extends beyond sequential operations to include random access file capabilities. With random access, programmers can read from or write to specific locations within a file, rather than solely operating sequentially from the beginning to the end.
- This functionality is particularly useful for large files or applications requiring frequent updates to specific sections of a file.
- By utilizing random access file techniques, developers can efficiently access and modify file content at precise locations, enhancing performance and flexibility in various applications, including databases, file editors, and data storage systems.

Keywords

File Stream Classes: File stream classes are programming constructs used to handle input and output operations on files in a structured and efficient manner. These classes typically provide functionalities for opening, reading from, writing to, and closing files. Examples include ifstream and ofstream in C++ and FileStream in C#. They allow programmers to interact with files as streams of data, enabling seamless integration of file operations into their applications.

Random Access File: A random access file is a type of file that allows data to be read from or written to any location within the file, as opposed to strictly sequential access from the beginning to the end. This means that data can be accessed non-sequentially, based on the byte offset or position within the file. Random access files are particularly useful for applications that require frequent updates or retrievals of specific data points within a file, such as databases, file editors, or data storage systems. They provide flexibility and efficiency in accessing and manipulating file content.

Input/Output (I/O): Input/Output, often abbreviated as I/O, refers to the communication between a computer and external devices, such as files, keyboards, displays, and networks. In the context of file handling, I/O operations involve reading data from files (input) and writing data to files (output). File I/O operations are essential for reading and writing data to and from files stored on a computer's storage system.

Sequential Access: Sequential access is a method of accessing data in a file by reading or writing it sequentially, starting from the beginning of the file and proceeding to the end in a linear manner. Each operation reads or writes data sequentially from the current position, advancing the file pointer to the next location after each operation. Sequential access is suitable for processing files where data is organized in a linear or ordered manner and does not require random access to specific locations within the file.

File Pointer: A file pointer, also known as a file position indicator, is a data structure maintained by the operating system or programming language runtime to keep track of the current position within a file during I/O operations. The file pointer indicates the location in the file where the next read or write operation will occur. As data is read from or written to the file, the file pointer is automatically updated to reflect the current position. The file pointer is essential for implementing sequential access and random-access operations on files.

Self Assessment

1) Question: Which of the following file stream classes is used for reading from a file in C++?

A. ifstream

B. ofstream

C. fstream

D. FileStream

2) Question: What does the acronym "I/O" stand for in the context of file handling?

A. Input/Output

B. Information/Observation

C. Interface/Operation

D. Interrupt/Override

3) Question: Which access method allows data to be read from or written to any location within a file?

A. Sequential Access

B. Random Access

C. Direct Access

D. Indexed Access

4) Question: In C++, which function is used to open a file for writing?

A. open()

B. write()

C. fopen()

D. ofstream::open()

5) Question: What is the purpose of the file pointer in file handling?

A. To indicate the current position within a file

B. To store the file's metadata

C. To manage file permissions

D. To encrypt file contents

6) Question: Which file access method involves reading or writing data in a linear manner from the beginning to the end of a file?

A. Sequential Access

B. Random Access

C. Direct Access

D. Indexed Access

7) Question: In C++, which header file must be included to work with file stream classes?

A. <iostream>

B. <fstream>

C. <cstdio>

D. <iostream> and <fstream>

8) Question: What does the ofstream class in C++ stand for?

A. OutputFileStream

B. OpenFileStream

C. OutputFile

D. OutputFileStream

9) Question: Which of the following operations is NOT typically performed using file handling?

A. Reading data from a database

B. Writing log files

C. Storing configuration settings

D. Processing user input

10) Question: Which method is used to close a file in C++ after performing file operations?

A. close()

B. end()

C. fclose()

D. finish()

11) Question: What is the default mode of opening a file in C++ if no mode is specified explicitly?

A. Read mode
B. Write mode
C. Append mode
D. Binary mode

12) Question: Which of the following is NOT a file access mode in C++?
A. ios::in
B. ios::out
C. ios::add
D. ios::binary

13) Question: Which function is used to check whether a file has been successfully opened in C++?
A. open()
B. is_open()
C. good()
D. fail()

14) Question: In C++, which operator is used for writing data to a file?
A. <<
B. >>
C. &
D. |

15) Question: What does the acronym "IOStream" stand for in C++?
A. InputOutputStream
B. InputOutputStream
C. InternalOutputStream
D. InputOnlyStream

Answers for Self Assessment

1.	A	2.	A	3.	B	4.	D	5.	A
6.	A	7.	D	8.	A	9.	D	10.	A
11.	A	12.	C	13.	B	14.	A	15.	A

Review Questions

1) Discuss the advantages and disadvantages of using sequential access compared to random access when working with large files.
2) Explain the difference between text files and binary files. Provide examples of situations where each type of file would be most appropriate.
3) Describe the process of error handling in file operations. How can programmers ensure robustness and reliability when dealing with file I/O errors?

4) Discuss the concept of buffering in file handling. How does buffering improve performance, and what are the potential drawbacks associated with buffering?
5) Explain the significance of file permissions in file handling. What types of permissions are typically available, and how do they impact file access and security?
6) Describe the role of the file system in managing files on a computer's storage system. How does the file system organize and store data, and what are its key components?

Further Readings

Balagurusamy E. Programming with Java 3e Primer. New Delhi: Tata McGraw Publishers.

Schildt H. Java A Beginner's Guide, 3rd ed. New York: Mc-Graw Hill.

Arnold, K., Gosling, J., & Holmes, D. (2005). *The Java programming language*. Addison Wesley Professional.

Haggar, P. (2000). *Practical Java: programming language guide*. Addison-Wesley Professional. Online

http://www.uop.edu.pk/ocontents/lec-14.pdf

https://www.scaler.com/topics/java/java-io-streams/

LOVELY PROFESSIONAL UNIVERSITY
Jalandhar-Delhi G.T. Road (NH-1)
Phagwara, Punjab (India)-144411
For Enquiry: +91-1824-521360
Fax.: +91-1824-506111
Email: odl@lpu.co.in