Module IV: Object Oriented Paradigm

Course: Object oriented Programming in JAVA

Dr. Vishwa Pratap Singh

NextGen Academy

February 10, 2024

NextGen Academy
Towards fulfilling a million dreams

Class Abstraction and Encapsulation

- Java follows OOPs concepts: classes, objects, polymorphism, inheritance.
- Abstraction hides unnecessary information, while encapsulation hides internal workings.
- **Abstraction:**
  - Feature of OOPs to hide unnecessary data.
  - Implemented using interfaces and abstract classes.
  - Example: TV remote abstraction.
- **Encapsulation:**
  - Binds data into a single unit (class).
  - Prevents external access to data members.
  - Enhances security.

Abstraction Example - Java Program

```
//abstract class
abstract class Shape {
  //abstract method
  public abstract void draw();
}

class Circle extends Shape {
  //implementing functionality
  public void draw() {
    System.out.println("Circle!");
  }
}

//main class
public class Test {
  public static void main(String[] args) {
    Shape circle = new Circle();
    circle.draw();
```

Encapsulation Example - Java Program

```
//Java class to test encapsulated class Account
public class EncapsulationDemo {
  public static void main(String[] args) {
    //creating instance of Account class
    Account acc = new Account();

    //setting values through setter methods
    acc.setAcc_no(7560504000L);
    acc.setName("Mark Dennis");
    acc.setEmail("md123@gmail.com");
    acc.setAmount(500000f);

    //getting values through getter methods
    System.out.println(acc.getAcc_no() + " " + acc.getName() + "
                       acc.getEmail() + " " + acc.getAmount());
  }
}
```

Encapsulation - Account Class

```
class Customer {
  //private data members
  private long acc_no;
  private String name, email;
  private float amount;

  //public getter and setter methods
  public long getAcc_no() {
    return acc_no;
  }
  // ... (similar methods for other fields)
}
```

Difference Between Abstraction and Encapsulation

Abstraction

- Hides unnecessary details.
- Solves issues at the design level.
- Focuses on external outlook.
- Implemented using abstract classes and interfaces.

Encapsulation

- Hides code and data into a single entity.
- Solves issues at the implementation level.
- Focuses on internal working.
- Implemented using access modifiers.

- **The Progress of Abstraction:**
  - All programming languages provide abstractions.
  - Complexity of problem-solving linked to abstraction quality.
- **An Object has an Interface:**
  - Objects represent elements in the problem space.
  - Class, instance, method, and messages in OOP.
- **An Object Provides Services:**
  - Think of objects as service providers.
  - Objects offer services to solve problems.

Thinking in Objects (contd.)

- **The Hidden Implementation:**
  - Class creators vs. client programmers.
  - Access control (public, private, protected) boundaries.
- **Reusing the Implementation:**
  - Code reuse is a significant advantage.
  - Reuse by using objects directly or composing new classes.
  - Composition (”has-a” relationship) enhances reusability.

NextGen Academy
Towards fulfilling a million dreams

Processing Primitive Data Types in Java

- Primitive data types in Java are predefined and named keywords.
- Eight primitive data types, including boolean, byte, int, long, float, double, char, and short.
- Each primitive type has a distinct purpose in storing data.

NextGen Academy
Towards fulfilling a million dreams

Primitive Number Types

- Whole numbers: byte, short, int, long.
- Floating numbers: float, double.
- Examples of whole and floating numbers.
- Most widely used types: int, double, float.

NextGen Academy
Towards fulfilling a million dreams

Data Types in Java

- Java is a statically-typed language.
- Variables must be declared before use.
- Two categories: Primitive and Non-primitive.
- Non-primitive types include class, interface, and Array.
- Primitive types example: `int a = 1;`

NextGen Academy
Towards fulfilling a million dreams

Boolean Data Type

- Represents true or false values.
- One bit of information.
- Example:

```
boolean isJtpBest=true; boolean isCold = false;
```

NextGen Academy
Towards fulfilling a million dreams

“begin–frame˝–Byte Data Type˝

“begin–itemize˝

“item 8bit signed 2′s complement integer.

“item Value range: 128 to 127.

“item Benefits of using byte data type.

“item Example:

“begin–verbatim˝

byte a = 100;

Int Data Type

- 32-bit signed two's complement integer.
- Value range: -2$^{3}1to(2^{3}1-1).Defaultvalueiszero$.
- Example:

```
int num= 50000;
```

“begin–frame˝–Long Data Type˝

“begin–itemize˝

“item 64bit 2′s complement integer.

“item Value range: 2ˆ63 to (2ˆ63 1).

“item Used for higher values.

“item Example:

“begin–verbatim˝

long l = 7000000000L;

Float Data Type

- Single-precision 32-bit IEEE 754 floating-point type.
- Useful for saving memory in large arrays.
- Recommended for large arrays of floating-point numbers.
- Example:

```
float num = 5.75f;
```

NextGen Academy
Towards fulfilling a million dreams

“begin–frame″–Double Data Type″

“begin–itemize″

“item Similar to float but with double precision.

“item Infinite value range.

“item Default value is 0.0d.

“item Example:

“begin–verbatim″

double num= 19.99d;

Char Data Type

- Essential for character values.
- Single 16-bit Unicode Character.
- Value range: 0 to 65,535 (inclusive).
- Example:

```
char myChar= 'H';
```

NextGen Academy
Towards fulfilling a million dreams

“begin–frame˝–Short Data Type˝

“begin–itemize˝

“item 16bit signed 2′s complement integer.

“item Value range: 32768 to 32767.

“item Used for memory saving.

“item Example:

“begin–verbatim˝

```
short num = 5000;
```

Default Values of Primitive Data Types

- Java assigns default values if not explicitly initialized.
- Default values for boolean, byte, int, long, float, double, char, short.
- Example:

```
int a; // Default value is 0
```

NextGen Academy
Towards fulfilling a million dreams

“begin–frame˝–Conversion to Wrapper Objects Example 1˝

“begin–itemize˝

“item Converting primitive types to wrapper objects.

“item Example program using Integer Double and Boolean wrappers.

“item Checking if objects belong to the corresponding wrapper class.

“end–itemize˝

“end–frame˝

“begin–frame˝–Conversion to Primitive Types Example 2˝

“begin–itemize˝

“item Converting wrapper objects to primitive types.

“item Example program using Integer Double and Boolean wrappers.

“item Printing primitive values.

“end–itemize˝

“end–frame˝

“begin–frame˝–The BigDecimal Class˝

“begin–itemize˝

“item BigDecimal provides precise arithmetic operations on double numbers.

“item Random precision integer unscaled value and a 32bit integer scale.

“item Example of BigDecimal operations.

“item Need for BigDecimal in handling floating point numbers.

“end–itemize˝

“end–frame˝

“begin–frame˝–BigDecimal Operations Example˝

“begin–itemize˝

“item Example program demonstrating BigDecimal operations.

“item Addition multiplication subtraction division power and negation.

“end–itemize˝

“end–frame˝

“begin–frame˝–Declaration and Initialization in BigDecimal˝

“begin–itemize˝

“item Declaring and initializing BigDecimal variables.

“item Using valueOf() new BigDecimal() and predefined constants.

“item Mathematical operations in BigDecimal.

“end–itemize˝

“end–frame˝

“begin–frame˝–BigInteger Class˝

“begin–itemize˝

“item Used for mathematical operations with very large integers.

“item Handy for competitive programming.

“item Initializing mathematical operations and extracting values in BigInteger.

“end–itemize”

“end–frame”

“begin–frame”–BigInteger Operations Example”

“begin–itemize”

“item Example program showcasing BigInteger operations.

“item Addition subtraction multiplication division and remainder.

“item Comparison of BigInteger values.

“end–itemize”

“end–frame”

“begin–frame”–Handling Large Factorials Example”

“begin–itemize”

“item Example program calculating factorial using BigInteger.

“item Handling large numbers beyond primitive data type limits.

“item Demonstrates the power of BigInteger in solving complex problems.

“end–itemize”

“end–frame”

“begin–frame”–Creating a String”

NextGen Academy
Towards fulfilling a million dreams

“begin–itemize˝

“item String literal: “texttt–String s = ”GeeksforGeeks”;˝

“item Using “texttt–new˝ keyword: “texttt–String s = new String(”GeeksforGeeks”);˝

“end–itemize˝

“end–frame˝

“begin–frame˝–String Constructors in Java˝

“begin–itemize˝

“item “texttt–String(byte[] byte“˙arr)˝

“item “texttt–String(byte[] byte“˙arr Charset char“˙set)˝

“item “texttt–String(byte[] byte“˙arr String char“˙set“˙name)˝

“item “texttt–String(byte[] byte“˙arr int start“˙index int length)˝

“item “texttt–String(byte[] byte“˙arr int start“˙index int length Charset char“˙set)˝

“item “texttt–String(byte[] byte“˙arr int start“˙index int length String char“˙set“˙name)˝

“item “texttt–String(char[] char“˙arr)˝

“item “texttt–String(char[] char“˙array int start“˙index int count)˝

“item “texttt–String(int[] uni“˙code“˙points int offset int count)˝

“item “texttt–String(StringBuffer s“˙buffer)˝

“end–itemize˝

“end–frame˝

“begin–frame˝–String Methods in Java˝

“begin–itemize˝

“item “texttt–int length()˝

“item “texttt–char charAt(int i)˝

“item “texttt–String substring(int i)˝

“item “texttt–String substring(int i int j)˝

“item “texttt–String concat(String str)˝

“item “texttt–int indexOf(String s)˝

“item “texttt–int indexOf(String s int i)˝

“item “texttt–int lastIndexOf(String s)˝

“item “texttt–boolean equals(Object otherObj)˝

“item “texttt–boolean equalsIgnoreCase(String anotherString)˝

“end–itemize˝

“end–frame˝

“section–String Builder and String Buffer˝

“begin–frame˝–String Builder and String Buffer Classes˝

“begin–itemize˝

“item String is immutable StringBuffer and StringBuilder are mutable.

“item Differences between StringBuffer and StringBuilder:

“begin–itemize”

“item StringBuffer is synchronized; StringBuilder is not.

“item StringBuffer is less efficient; StringBuilder is more efficient.

“item StringBuffer introduced in Java 1.0; StringBuilder introduced in Java 1.5.

“end–itemize”

“end–itemize”

“end–frame”

“begin–frame”–StringBuffer Example”

“begin–itemize”

“item “texttt–BufferTest.java”

“begin–verbatim”

```
public class BufferTest –
public static void main(String[] args) –
StringBuffer buffer = new StringBuffer(”hello”);
buffer.append(”java”);
System.out.println(buffer);
”
```

”

Output: `hellojava`

StringBuilder Example

- `BuilderTest.java`

```
public class BuilderTest  public static void main(String[] args)
StringBuilder builder = new StringBuilder("hello"); builder.appen
System.out.println(builder);
```

- **Output:** `hellojava`

“begin–frame˝–Performance Test of StringBuffer and StringBuilder˝

“begin–itemize˝

“item “texttt–ConcatTest.java˝

“begin–verbatim˝

public class ConcatTest –

public static void main(String[] args) –

// ... (code omitted for brevity)

˝

˝

Output: Time taken by StringBuffer: 16ms, Time taken by StringBuilder: 0ms

Method Overloading Example

static int add(int a, int b, int c) return a + b + c;

// Main Function public static void main(String args[]) System.out.println(”add() with 2 parameters”); System.out.println(add(4, 6)); System.out.println(”add() with 3 parameters”); System.out.println(add(4, 6, 7));

Method Overriding Example

oid eat() System.out.println(”eat() method of base class”); System.out.println(”eating.”);

class Dog extends Animal void eat() System.out.println(”eat() method of derived class”); System.out.println(”Dog is eating.”);

The Object Class and Its `toString()` Method

- Java as a true object-oriented programming language.
- Designing applications using classes and objects.
- All Java programs composed of classes.
- Class as a model to create objects with properties and actions.
- Object encapsulates state and behavior.
- Declaration of class and its components.
- Real-world examples of objects and classes.
- Flow of execution in a Java program.
- Simple programming structure of Java classes and objects.
- Difference between class and object.

NextGen Academy
Towards fulfilling a million dreams

What is an Object in Java

- Object as a basic unit of object-oriented programming.
- Real-world entities with properties and actions.
- State and behavior of an object.
- Examples: book, pen, pencil, etc.
- Objects consist of attributes (data members) and behaviors (methods).
- Object as an instance of a class.

NextGen Academy
Towards fulfilling a million dreams

Characteristics of Object in Java

- State, behavior, and identity of an object.
- State represented by instance variables.
- Behavior represented by methods.
- Identity as a unique name of an object.
- Real-time examples illustrating characteristics.

What is a Class in Java

- Class as a fundamental building block of OOP.
- Blueprint/template of an object.
- Contains objects with similar states and behaviors.
- Real-time examples of classes.
- Types of classes in Java.

NextGen Academy
Towards fulfilling a million dreams

Class Declaration in Java

- Declaration using `class` keyword.
- Syntax: `modifierName class className { // class body. }`
- Components of a class.
- Modifiers, class name, and body.

NextGen Academy
Towards fulfilling a million dreams

Components of Class in Java

- Modifiers: public, private, default, protected.
- Class name conventions.
- Body enclosed in braces.
- Members: Fields, Constructors, Methods, Blocks.
- Introduction to interfaces and `main` method.

NextGen Academy
Towards fulfilling a million dreams

Flow of Execution of Program in Java

- Execution sequence: static variables, blocks, methods.
- Instance variable execution during object creation.
- Execution of instance block before constructor.
- Order of execution: static to instance, constructor, method, local variable.
- Example illustrating the flow.

Simple Programming Structure of Class and Objects in Java

- Example: `Student` class.
- Declaration of state/properties.
- Declaration of constructor and actions.
- Introduction to object instantiation.
- Key points about classes and objects.

NextGen Academy
Towards fulfilling a million dreams

Key Points concerning Classes and Objects in Java

- Fundamental terms in OOP: Objects, classes, abstraction, encapsulation, etc.
- Objects as basic runtime entities.
- Class representing a group of similar objects.
- Object creation and instances.
- Members of a class: variables and methods.

NextGen Academy
Towards fulfilling a million dreams

Polymorphism

- Core concept in OOP.
- Derived from Greek words: poly (many) and morphs (forms).
- Achieving a single task in different ways.
- Flexibility in code using polymorphism.

NextGen Academy
Towards fulfilling a million dreams

Realtime Example of Polymorphism in Java

- Examples in the world: water changing states, human behavior, etc.
- Polymorphism in human behavior.
- Single button for computer ON and OFF.
- Love example showcasing polymorphism.

Types of Polymorphism in Java

- Static polymorphism.
- Dynamic polymorphism.

Static Polymorphism in Java

- Exhibited during compilation.
- Behavior decided at compile-time.
- Achieved through method overloading.
- Example program demonstrating static polymorphism.

NextGen Academy
Towards fulfilling a million dreams

Java Compile-time Polymorphism Example Program

```
package staticPolymorphism;

public class StaticPoly {
    void sum(int x, int y) {
        int s = x + y;
        System.out.println("Sum of two numbers: " + s);
    }

    void sum(int x, int y, int z) {
        int s = x + y + z;
        System.out.println("Sum of three numbers: " + s);
    }

    public static void main(String[] args) {
        StaticPoly obj = new StaticPoly();
        obj.sum(20, 10);
        obj.sum(10, 20, 30);
    }
```

Conclusion

- Recap of key concepts covered.
- Importance of objects, classes, and polymorphism in Java.
- Building a strong foundation for OOP in Java.

NextGen Academy

Towards fulfilling a million dreams

Dynamic Binding

- Dynamic binding in Java occurs during runtime.
- Resolved based on the type of object at runtime.
- Also known as late binding or runtime binding.
- Example: Method overriding.
- JVM resolves method calls based on the object type.
- Binding happens at runtime, not compile time.
- Code example provided for clarification.
- Object type is determined at runtime.
- JVM resolves method calls dynamically.
- Dynamic polymorphism demonstrated.

Dynamic Binding (Contd.)

- Example program with Animal and Lion classes.
- Output explanation provided for clarity.
- Compiler cannot determine the object type.
- JVM resolves method calls at runtime.
- Assignment reference determines method call.
- Importance of dynamic binding in Java.
- Understanding method invocation dynamically.
- Addressing challenges in object type determination.
- Utilization of dynamic polymorphism.

NextGen Academy
Towards fulfilling a million dreams

Casting Objects and the instanceof Operator

- Typecasting in Java: upcasting and downcasting.
- Upcasting: Subtype to superclass, automatic procedure.
- Implicit and explicit casting explained.
- Downcasting: Subclass type refers to the parent class object.
- Class type casting rules provided.
- Implementation of upcasting and downcasting.
- Example demonstrating upcasting.
- Handling downcasting using the instanceof operator.
- ClassCastException and its runtime nature.

NextGen Academy
Towards fulfilling a million dreams

Casting Objects and the instanceof Operator (Contd.)

- Example illustrating downcasting and instanceof usage.
- Importance of ”IS-A-Relationship” in class casting.
- Upcasted object’s property for successful downcasting.
- Practical implementation of class type casting.
- Exception handling using instanceof.

NextGen Academy
Towards fulfilling a million dreams

What Is the instanceof Operator?

- instanceof: Binary operator for type testing.
- Checks if an object is of a given type.
- True or false result.
- Type comparison operator.
- Example with Round and Ring classes.
- Basic syntax: `(object) instanceof (type)`.
- Importance of instanceof in type checking.
- Avoiding ClassCastException using instanceof.
- Ensuring safe casting before runtime.

NextGen Academy
Towards fulfilling a million dreams

How Does the instanceof Operator Work?

- instanceof based on the is-a relationship.
- Demonstrated with Shape interface and Circle class.
- True if object is instance of type or its subclass.
- instanceof and interface implementation.
- instanceof limitations without a relationship.
- Example with Circle and Triangle classes.
- Compilation error without a relationship.
- Understanding is-a relationship for instanceof.
- Checking type compatibility for instanceof.

Using instanceof With the Object Type

- Every class implicitly inherits from Object class.
- instanceof with Object type always true.
- Demonstrated with Thread class.
- Importance of Object class inheritance.
- Object type and its relevance in instanceof.
- Ensuring compatibility with Object type.
- Utilization of instanceof for Object type check.

Using instanceof Operator When an Object Is null

- instanceof on null object returns false.
- Example with Circle object set to null.
- No need for null check with instanceof.
- Ensuring safe usage of instanceof with null.
- Practical demonstration of null check.

NextGen Academy
Towards fulfilling a million dreams

instanceof and Generics

- Instance tests and casts depend on runtime type information.
- instanceof limitations with erased generic types.
- Reified types in Java and instanceof.
- Compilation error example with generics.
- Types allowed with instanceof in Java.
- Handling generics with instanceof.
- Ensuring reified types for instanceof.

NextGen Academy
Towards fulfilling a million dreams

Conclusion

- Covered key Java concepts: Dynamic Binding, Casting Objects, and instanceof Operator.
- Understanding dynamic polymorphism and type casting.
- Importance of instanceof in safe type checking.
- Addressed challenges in upcasting and downcasting.
- Demonstrated practical examples for better comprehension.

NextGen Academy
Towards fulfilling a million dreams

The Object's equals Method

- `equals(Object obj)` method of Object class.
- Used to compare objects based on their content.
- Suggested to override for custom equality conditions.
- **Syntax:** `public boolean equals(Object obj)`
- **Parameter:** `obj` - reference object to be compared.
- **Returns:** `true` if objects are equal, `false` otherwise.
- Example code:

```
@Override public boolean equals(Object obj)  // Custom equality
condition
```

NextGen Academy
Towards fulfilling a million dreams

The Object's equals Method (Contd.)

- Example code demonstrated for using `equals` method.
- Output explained for better comprehension.

The Array List Class

- `ArrayList` class in Java for dynamic arrays.
- No size limit, more flexible than traditional arrays.
- Implements List interface, maintains insertion order.
- Key points about `ArrayList` class:
- Can contain duplicate elements.
- Maintains insertion order.
- Non-synchronized.

NextGen Academy
Towards fulfilling a million dreams

The Array List Class (Contd.)

- `ArrayList` allows random access based on index.
- Manipulation is slower compared to LinkedList due to shifting.
- Cannot create `ArrayList` of primitive types directly.
- Example syntax for creating `ArrayList`:

```
ArrayList<Integer> list = new ArrayList<>();
```

NextGen Academy
Towards fulfilling a million dreams

% Slide 5

“begin–frame˝–Hierarchy of ArrayList class˝

“begin–itemize˝

“item “texttt–ArrayList˝ extends “texttt–AbstractList˝ and implements “texttt–List˝.

“item “texttt–List˝ extends “texttt–Collection˝ and “texttt–Iterable˝ interfaces.

“end–itemize˝

“end–frame˝

% Slide 6

“begin–frame˝–ArrayList class declaration˝

“begin–itemize˝

“item Declaration of “texttt–ArrayList˝ class with generics:

“end–itemize˝

“begin–verbatim˝

public class ArrayListE extends AbstractListE

implements ListE RandomAccess Cloneable Serializable

Constructors of ArrayList

- `ArrayList()` - Builds an empty array list.
- `ArrayList(Collection<? extends E> c)` - Initializes with elements of collection `c`.
- `ArrayList(int capacity)` - Builds with specified initial capacity.

NextGen Academy
Towards fulfilling a million dreams

Java Non-generic Vs. Generic Collection

- Evolution from non-generic to generic collection in Java.
- Type safety in generic collections.
- Comparison of old and new ways of creating `ArrayList`.

NextGen Academy
Towards fulfilling a million dreams

Java Non-generic Vs. Generic Collection (Contd.)

- Type specification in angular braces for generic collections.
- Compile-time error for attempting to add a different type.

NextGen Academy
Towards fulfilling a million dreams

Java ArrayList Example

- Example code demonstrating the use of `ArrayList` in Java.
- Output explained for the provided code.

NextGen Academy

Towards fulfilling a million dreams