Assignment-1: Java Basics (10 June 2024)

Class

A class in Java is like a blueprint for creating objects. It defines the properties (variables) and behaviors (methods) that the objects created from the class will have. For example, a Car class might have properties like color and model, and behaviors like start and stop.

```
public class Car {
   String color;
   String model;
   void start() {
      System.out.println("Car started");
   }
   void stop() {
      System.out.println("Car stopped");
   }
}
```

Methods

Methods are blocks of code that perform a specific task. They are like functions that belong to a class. Methods define the behavior of the objects created from the class.

```
public class Calculator {
   int add(int a, int b) {
      return a + b;
   }
   void displayMessage() {
      System.out.println("Hello, world!");
   }
}
```

Access Modifiers

Access modifiers in Java control the visibility of classes, methods, and variables. They define how accessible these elements are from other parts of the program.

1. **public**: Accessible from anywhere.
2. **private**: Accessible only within the same class.
3. **protected**: Accessible within the same package and subclasses.
4. **default** (no modifier): Accessible only within the same package.

```
public class Example {
   public int publicVar;
   private int privateVar;
   protected int protectedVar;
   int defaultVar; // default access modifier
}
```

Package

A package in Java is a way to group related classes and interfaces together. It helps organize code and manage the namespace by preventing naming conflicts. Packages are like folders in a file system.

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

Java Assignment-2: Keywords (11 June 2024)

Access Modifiers

- **private**: Restricts access to members (fields, methods) within the same class only. Its scope is within the class.
- **protected**: Allows access to members within the same package and subclasses. Its scope is within the package and outside the package's child class.
- **public**: Makes members accessible from any other class. It can be accessible anywhere.
- **default:** It allows the access of the data-members within the package.

Class, Interface, and Object Keywords

- **class**: Declares a class.
- **interface**: Declares an interface.
- **enum**: Declares an enumerated type.
- **extends**: Indicates that a class is inheriting from a superclass.
- **implements**: Indicates that a class implements an interface.
- **new**: Creates new objects.

Control Flow Statements

a. **if**: Conditional statement, executes code block if the condition is true.

b. **else**: Executes alternative code block if the preceding if condition is false.

c. **switch**: Selects one of many code blocks to execute.

d. **case**: Defines a path in a switch statement.

e. **default**: Specifies the default block of code in a switch statement if no case matches.

f. **while**: Repeatedly executes a block of statements while a condition is true.

g. **do**: Executes a block of statements once, then repeats while a condition is true.

h. **for**: Iterates over a range or collection.

i. **break**: Exits from the current loop or switch statement.

j. **continue**: Skips the current iteration of a loop and proceeds to the next iteration.

k. **return**: Exits from the current method and optionally returns a value.

l. **try**: Starts a block of code that will be tested for exceptions.

m. **catch**: Catches exceptions thrown by the try block.

n. **finally**: Executes a block of code after the try and catch blocks, regardless of whether an exception was thrown or not.

o. **throw**: Throws an exception.

p. **throws**: Declares exceptions that a method might throw.

Primitive Data Types

- **byte**: 8-bit integer data type.
- **short**: 16-bit integer data type.
- **int**: 32-bit integer data type.
- **long**: 64-bit integer data type.
- **float**: Single-precision 32-bit floating point.
- **double**: Double-precision 64-bit floating point.
- **char**: Single 16-bit Unicode character.
- **boolean**: Data type with only two possible values: true and false.

Non-Access Modifiers

- **abstract**: Specifies that a class or method is abstract and cannot be instantiated or must be implemented by subclasses.
- **final**: Prevents further inheritance or modification of a class, method, or variable.
- **static**: Indicates that a member belongs to the class, rather than instances of the class.
- **synchronized**: Ensures that a method or block of code is only accessed by one thread at a time.
- **volatile**: Indicates that a variable's value will be modified by different threads.
- **transient**: Prevents serialization of a field.

Miscellaneous

- **this**: Refers to the current instance of a class.
- **super**: Refers to the superclass of the current object.
- **void**: Specifies that a method does not return a value.

- **const**: Reserved but not used.
- **goto**: Reserved but not used.
- **instanceof**: Tests whether an object is an instance of a specific class or interface.
- **package**: Declares a package.
- **import**: Imports other Java packages or classes.
- **native**: Indicates that a method is implemented in native code using JNI (Java Native Interface).
- **strictfp**: Restricts floating-point calculations to ensure portability.
- **null**: Represents a null reference.
- **assert**: Used for debugging purposes to make an assertion.

Java Assignment-3: OOP's (12 June 2024)

1. Java Inheritance

Inheritance in Java allows one class to inherit the fields and methods of another, promoting code reuse and establishing a natural hierarchical relationship between classes. The extends keyword is used to inherit from a parent class.

```
class ParentClass {
   // fields and methods
}
class ChildClass extends ParentClass {
   // additional fields and methods
}
```

2. Polymorphism

Polymorphism in Java allows objects to be treated as instances of their parent class rather than their actual class, enabling a single interface to represent different underlying forms (data types). This is achieved through method overriding and method overloading.

```
class ParentClass {
   void display() {
      System.out.println("Parent display");
   }
}

class ChildClass extends ParentClass {
   void display() {
      System.out.println("Child display");
   }
}
```

```
public class TestPolymorphism {
    public static void main(String[] args) {
        ParentClass obj = new ChildClass();
        obj.display();  // Outputs: Child display
    }
}
```

3. Method Overloading in Java

Method overloading in Java occurs when multiple methods in the same class have the same name but different parameters (different type, number, or both). It allows a class to perform a similar operation in different ways, depending on the method signature.

```
class MyClass {
    void display(int a) {
        System.out.println("Argument: " + a);
    }
    void display(String a) {
        System.out.println("Argument: " + a);
    }
    void display(int a, String b) {
        System.out.println("Arguments: " + a + ", " + b);
    }
}
```

4. Method Overriding in Java

Method overriding in Java happens when a subclass provides a specific implementation for a method already defined in its superclass. The method in the subclass must have the same name, return type, and parameters as the one in the superclass.

```
class ParentClass {
    void display() {
        System.out.println("Parent display");
    }
}
class ChildClass extends ParentClass {
    @Override
    void display() {
        System.out.println("Child display");
    }
}
```

5. Java Abstraction

Abstraction in Java is the concept of hiding the complex implementation details and showing only the necessary features of an object. It can be achieved using abstract classes and interfaces.

```
abstract class AbstractClass {
    abstract void abstractMethod();
    void concreteMethod() {
        System.out.println("Concrete method");
    }
}
class ConcreteClass extends AbstractClass {
    @Override
    void abstractMethod() {
        System.out.println("Abstract method implementation");
    }
}
```

```
interface MyInterface {
    void abstractMethod();
}

class ImplementingClass implements MyInterface {
    @Override
    public void abstractMethod() {
        System.out.println("Interface method implementation");
    }
}
```

6. Java Encapsulation

Encapsulation in Java is the practice of bundling the data (fields) and methods (functions) that operate on the data into a single unit or class, and restricting access to the internal state through public methods. This is achieved using private fields and public getter and setter methods.

```
class EncapsulatedClass {
    private int field;
    public int getField() {
        return field;
    }
    public void setField(int field) {
        this.field = field;
    }
}
```

7. Rules for Java Method Overriding

7.1) The method in the child class must have the same name, return type, and parameters as in the parent class.

7.2) The overriding method cannot have more restrictive access than the overridden method, and it can throw only those exceptions declared in the parent class or subclasses of those exceptions.

8. Constructors in Java

Constructors in Java are special methods that are called when an object is instantiated. They initialize the new object and can be overloaded to provide different ways of initializing the object.

```
class MyClass {
   // Default constructor
   public MyClass() {
      System.out.println("Default constructor");
   }
   // Parameterized constructor
   public MyClass(int param) {
      System.out.println("Parameterized constructor with param: " + param);
   }
}
```

9. Types of Java Constructors

9.1) **Default Constructor:** A constructor with no parameters, automatically provided by Java if no other constructor is defined.

```
class MyClass {
   public MyClass() {
      System.out.println("Default constructor");
   }
}
```

9.2) **Parameterized Constructor:** A constructor that accepts arguments to initialize an object with specific values.

```
class MyClass {
   public MyClass(int param) {
      System.out.println("Parameterized constructor with param: " + param);
   }
}
```

10. Constructor Overloading in Java

Constructor overloading in Java occurs when a class has more than one constructor, each with different parameter lists. This allows for creating objects in different ways, depending on the provided parameters.

```
class MyClass {
   public MyClass() {
      System.out.println("Default constructor");
   }
   public MyClass(int param) {
      System.out.println("Parameterized constructor with param: " + param);
   }
   public MyClass(int param1, String param2) {
      System.out.println("Constructor with params: " + param1 + ", " + param2);
   }
}
```

11. Interface in Java

An interface in Java is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces allow for multiple inheritance and define a contract that classes can implement.

```
// Define the interface
interface Animal {
    void makeSound();
}
// Implement the interface in a class
class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof");
    }
}
// Test the interface implementation
public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.makeSound(); // Outputs: Woof
    }
}
```

Summary:

In this guide, I have provided a concise and clear explanation of key Object-Oriented Programming (OOP) concepts in Java. Each concept is accompanied by straightforward syntax and simple examples to illustrate the ideas effectively and make them easy to grasp. This approach ensures that the fundamentals of OOP are accessible and understandable for anyone looking to learn or refresh their knowledge

Java Assignment-4: (13 June 2024)

Exception Handling & Java Framework

1) Exception Handling:

Hierarchy of Exception Classes

The root class for exceptions in Java is Throw able, which has two main subclasses: Exception (for recoverable conditions) and Error (for serious problems). Exception is further divided into checked exceptions (must be declared or handled) and unchecked exceptions (runtime exceptions).

Types of Exception

- **Checked Exceptions:** Must be declared in a method or constructor's throws clause if they can be thrown by the execution of the method or constructor and propagated outside the method or constructor boundary.
- **Unchecked Exceptions:** Include Runtime Exception and its subclasses; these are not checked at compile time.

try: The try block contains code that might throw an exception and is used to catch and handle exceptions.

catch: The catch block handles the exception that occurs in the associated try block.

finally: The finally block contains code that is always executed after the try and catch blocks, regardless of whether an exception was thrown or not.

throw: The throw statement is used to explicitly throw an exception in the program.

throws: The throws keyword is used in method signatures to declare that a method might throw one or more exceptions.

Multiple catch block: Multiple catch blocks can be used to handle different types of exceptions separately for the same try block.

Exception Handling with Method Overriding: When overriding a method, the overriding method can only throw exceptions that are the same or subclasses of the exceptions declared in the parent class method.

2) Java Collection Framework:

Hierarchy of Collection Framework

The Java Collection Framework consists of interfaces and classes that implement those interfaces. The main interfaces are Collection, List, Set, Queue, and Map, each having multiple implementations.

Collection interface

The Collection interface is the root of the collection hierarchy and represents a group of objects known as elements. It includes methods for adding, removing, and querying elements.

Iterator interface

The Iterator interface provides methods to iterate over any collection, such as hasNext(), next(), and remove().

Set, List, Queue, Map, Comparator, Comparable interfaces

- **Set:** A collection that does not allow duplicate elements.
- **List:** An ordered collection that allows duplicates.
- **Queue:** A collection used to hold multiple elements prior to processing, typically in FIFO order.
- **Map:** A collection that maps keys to values, with no duplicate keys allowed.
- **Comparator:** Used to define custom ordering for objects.
- **Comparable:** Used to define the natural ordering of objects.

ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet, HashMap, ConcurrentHashMap classes

- **ArrayList:** A resizable array implementation of the List interface.
- **Vector:** Similar to ArrayList, but synchronized.
- **LinkedList:** Doubly linked list implementation of the List and Deque interfaces.
- **PriorityQueue:** A queue that orders elements according to their natural ordering or a Comparator.
- **HashSet:** A set based on a hash table, does not maintain order.

- **LinkedHashSet:** A hash table and linked list implementation of the Set interface, maintains insertion order.
- **TreeSet:** A NavigableSet implementation based on a red-black tree, orders elements naturally or by a Comparator.
- **HashMap:** A map based on a hash table, does not maintain order.
- **ConcurrentHashMap:** A thread-safe implementation of the Map interface.

Java Assignment -5: Multithreading (14 June 2024)

Thread Life Cycle

A thread in a programming environment goes through several states during its life:

1. **New**: The thread is created but not yet started.
2. **Runnable**: The thread is ready to run and waiting for CPU time.
3. **Running**: The thread is actively executing.
4. **Dead**: The thread has finished execution or has been terminated.

Multithreading

Multithreading is the ability of a CPU to execute multiple threads concurrently, which allows for better utilization of resources and improved performance of applications by performing multiple operations at once.

Synchronization

Synchronization is a mechanism that ensures that two or more concurrent threads do not simultaneously execute some particular program segment known as a critical section. This is crucial to avoid conflicts and ensure data consistency when multiple threads access shared resources.

Deadlock

A deadlock occurs when two or more threads are blocked forever, each waiting for the other to release a resource. This typically happens in a situation where each thread holds a lock and waits for another lock held by another thread, creating a cycle of dependencies with no resolution.

Executor Framework

The Executor Framework in Java provides a higher-level replacement for managing threads. It includes different types of thread pools for managing and executing tasks:

1. **Fixed Thread Pool**:
    - A pool with a fixed number of threads.
    - Useful for managing a known number of concurrent tasks.
2. **Single Thread Pool**:
    - A pool with only one thread.
    - Ensures that tasks are executed sequentially, one at a time.
3. **Cached Thread Pool**:
    - A dynamically sized pool that creates new threads as needed but reuses previously constructed threads when they are available.
    - Suitable for executing many short-lived tasks.
4. **Scheduled Thread Pool**:
    - A pool that can schedule commands to run after a given delay, or to execute periodically.
    - Useful for tasks that need to run at regular intervals or after a delay.