Object Oriented Programming - Java

30/08/2023

Jumps statements

- continue:
  - skips the current iteration of a loop
- break:
  - breaks out of the loop of the parent loop
  - exit a switch case
  - as a goto statement

31/08/2023

Operators

- Arithmetic
  - +, -, *, /, %, ++, --
- Relational
  - ==, !=, >, <, >=, <=
- Logical
  - &&, ||, !
- Bitwise
  - &, |, ^, ~, <<, >>
- Assignment
  - =, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=
- Misc
  - sizeof(), &, *, ?:, ., ->, (type), [], (type){}, (type)()
- Shift
  - <<, >>, >>>
- Unary
  - +, -, ++, --, !
- Ternary

    - ?:

- Compound Assignment
    - +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=

- Precedence
    - ()
    - ++, --, +, -, !
    - *, /, %
    - +, -
    - <<, >>, >>>
    - <, <=, >, >=
    - ==, !=
    - &
    - ^
    - |
    - &&
    - ||
    - ?:
    - =, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=

02/09/2023

Arrays

- Collection of similar data types in a contiguous memory location
- Arrays are objects of dynamically generated classes
- Arrays are of fixed size
- Every element in an array is of the same type and just a variable
- Declaration:
    - `int[] arr;`
    - `int arr[];`
    - `int[] arr = new int[5];`
    - `int[] arr = {1, 2, 3, 4, 5};`
- Instantiation:
    - `arr = new int[5];`
    - `arr = {1, 2, 3, 4, 5};`
- Initialization:
    - `arr[0] = 1;`
    - `arr[1] = 2;`
    - `arr[2] = 3;`
    - `arr[3] = 4;`
    - `arr[4] = 5;`
- Default values:
    - int: 0
    - float: 0.0

    - boolean: false
    - char: '\u0000'
    - String: null

Multidiemensional arrays:

- Array of arrays, can be 2D, 3D, 4D, etc.
- Stored in tabular form(row major order)
- Declaration:
    - `int[][] arr = new int[2][3];`
    - `int[][] arr = {{1, 2, 3}, {4, 5, 6}};`
- Instantiation:
    - `arr = new int[2][3];`
    - `arr = {{1, 2, 3}, {4, 5, 6}};`
- Initialization:
    - `arr[0][0] = 1;`
    - `arr[0][1] = 2;`
    - `arr[0][2] = 3;`
    - `arr[1][0] = 4;`
    - `arr[1][1] = 5;`
    - `arr[1][2] = 6;`
- Three dimensional array:
    - `int[][][] arr = new int[2][3][4];`
    - `int[][][] arr = {{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}}, {{13, 14, 15, 16}, {17, 18, 19, 20}, {21, 22, 23, 24}}};`
- Jagged array:
    - `int[][] arr = new int[2][];`
    - `int[][] arr = {{1, 2, 3}, {4, 5}};`
    - `int[][] arr = new int[2][3];`
    - `arr[0] = new int[3];`
    - `arr[1] = new int[2];`
    - `arr[0][0] = 1;`
    - `arr[0][1] = 2;`
    - `arr[0][2] = 3;`
    - `arr[1][0] = 4;`
    - `arr[1][1] = 5;`
- Indexing:
    - Accessed using i and j
    - i is the row number
    - j is the column number
- Length:
    - `arr.length` gives the number of rows
    - `arr[i].length` gives the number of columns in the ith row

Advantages:

- Code optimization:

    - It makes the code optimized, we can retrieve or sort the data easily.
- Random access:
    - We can get any data located at an index position.
- Utilize memory efficiently:
    - It reduces the memory space waste because arrays can be created in any data type.

Disadvantages:

- Size Limit:
    - We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.
- Cost of insertion and deletion:
    - The cost of insertion and deletion is quite high because the elements are stored in consecutive memory locations and the shifting operation is costly. If you want to insert a new element in an array then you need to shift the existing element that is why the cost is high.

cloning? copying?

03/09/2023

Classes and Objects

Class:

- A class is a blueprint for the object
- It is a template or a set of instructions to build a specific type of object
- It is a user defined data type
- It is a logical entity
- It is a reference type
- It is a collection of objects
- It is a group of similar objects
- It is a group of similar actions
- It is a group of similar states
- States and behaviors are represented by variables and methods respectively
- SYntax:
    - `Modifier class ClassName { // variables // methods }`
- Example:
    - `public class Student { int rollNo; String name; String address; public void display() { System.out.println("Roll no: " + rollNo);`

```
System.out.println("Name: " + name); System.out.println("Address: " +
address); } }
```

- Access modifiers:
    - public: accessible everywhere
    - private: accessible only within the class
    - protected: accessible within the package and outside the package but through inheritance only
    - default: accessible within the package only
      Access modifiers specify the accessiblity and permissions of a class, method, constructor, variable, etc.
- body of a class is defined within curly braces
- super class is the class from which the class inherits the properties and methods
- sub class is the class that inherits the properties and methods from the super class
- built-in classes:
  examples:
    - java.Lang.String
    - java.Lang.System
    - java.util.Scanner
      etc
- user defined:
    1. concrete
    2. abstract
    3. interface

Rules for creating classes:

- class name should start with a capital letter
- class name should not contain spaces
- class name should not contain special characters except underscore and dollar sign
- class name should not be a keyword
- can contain multpile classes but only one public class
- can inherit from only one class

Objects

- An object is an instance of a class
- Includes real world entities and maniuplates them using methods
- consists of:
    - state: variables
    - behavior: methods
    - identity: name

- Syntax:
  - `ClassName objectName = new ClassName();`
- Example:
  - `Student s1 = new Student();`
- 3 stages of object creation:
  1. declaration:
     - `Student s1;`
  2. instantiation
     - `s1 = new Student();`
  3. initialization
     - `s1.rollNo = 1;`
     - `s1.name = "John";`
     - `s1.address = "New York";`

todo: write a program with 1 class and multpile objects

12/09/2023

Declaring objects

- Syntax:
  - `ClassName objectName = new ClassName();`
- Example:
  - `Student s1 = new Student();`
- 3 stages of object creation:
  1. declaration:
     - `Student s1;`
  2. instantiation
     - `s1 = new Student();`
  3. initialization
     - `s1.rollNo = 1;`
     - `s1.name = "John";`
     - `s1.address = "New York";`
- `new` operator
  - used to create an object
  - allocates memory for the object
  - returns the reference of the object

Methods

- A method is a block of code that performs a specific task
- Syntax:
  - `accessModifier returnType methodName(parameterList) { // body return returnType; }`
- Example:
  - `public int add(int a, int b) { return a + b; }`

- Access modifiers:
  - public: accessible everywhere
  - private: accessible only within the class
  - protected: accessible within the package and outside the package but through inheritance only
  - default: accessible within the package only
- return type:
  - void: no return value
  - int: returns an integer
  - float: returns a float
  - double: returns a double
  - char: returns a character
  - boolean: returns a boolean
  - String: returns a string
  - ClassName: returns an object of the class
- parameter list:
  - list of parameters
  - parameters are variables that are passed to the method
  - parameters are optional
  - parameters are separated by commas
  - parameters are specified after the method name
  - parameters are specified within parentheses
  - parameters are specified as type followed by name
  - parameters are passed by value

calling a method:

- `objectName.methodName();`
- `objectName.methodName(parameterList);`
- When a method is called from a particular object, the version of the method that is called depends on the type of the object.

Constructors

- A constructor is a special method that is used to initialize objects
- It is automatically called when an object of a class is created
- It has the same name as the class
- It has no return type, default return type of class constructor is the class itself
- Syntax:

```
accessModifier ClassName(parameterList) {
    // body
}
```

- Example:

```
public Student(int rollNo, String name, String address) {
    this.rollNo = rollNo;
    this.name = name;
    this.address = address;
}
```

Method overloading:

- Method overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different.
- It is used to add more to the behavior of methods
- It is used to reuse the code
- Example:

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

- Rules:
  - The return type of the methods can be same or different, but the parameters should be different.
  - The parameters can be different in terms of:
    - Number of parameters
    - Data type of parameters
    - Sequence of data type of parameters
  - The access modifier can be same or different.
  - The method name should be same.

Constructor overloading:

- Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter list.

- It is used to add more to the behavior of constructors
- It is used to reuse the code
- Example:

```
public Student() {
    rollNo = 1;
    name = "John";
    address = "New York";
}
public Student(int rollNo, String name, String address) {
    this.rollNo = rollNo;
    this.name = name;
    this.address = address;
}
public Student(int rollNo, String name) {
    this.rollNo = rollNo;
    this.name = name;
    address = "New York";
}
public Student(int rollNo) {
    this.rollNo = rollNo;
    name = "John";
    address = "New York";
}
```

- Rules:
    - The return type of the methods can be same or different, but the parameters should be different.
    - The parameters can be different in terms of:
        - Number of parameters
        - Data type of parameters
        - Sequence of data type of parameters
    - The access modifier can be same or different.
    - The method name should be same.

20/09/2023

Inheritance

- Inheritance is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- It is used to achieve runtime polymorphism
- It is used to reuse the code
- It is used to add more to the behavior of classes

- Is-a relationship. Example: Car is a vehicle. Car is a child class of Vehicle. Vehicle is a parent class of Car.
- Terminoogy:
    - Super class: parent class
    - Sub class: child class
    - Base class: parent class
    - Derived class: child class
- Syntax:

```
class SuperClass {
    // body
}
class SubClass extends SuperClass {
    // body
}
```

- Example:

```
class Vehicle {
    String color;
    int maxSpeed;
    public void display() {
        System.out.println("Color: " + color);
        System.out.println("Max speed: " + maxSpeed);
    }
}
class Car extends Vehicle {
    int numGears;
    boolean isConvertible;
    public void display() {
        super.display();
        System.out.println("Number of gears: " + numGears);
        System.out.println("Is convertible: " + isConvertible);
    }
}
```

- Types of inheritance:
    - Single inheritance:
        - one parent class and one child class
        - Example: A is a parent class, B is a child class of A
    - Multilevel inheritance:
        - one parent class and multiple child classes
        - Example: A is a parent class, B is a child class of A, C is a child class of B

    - Hierarchical inheritance:
        - one parent class and multiple child classes
        - Example: A is a parent class, B is a child class of A, C is a child class of A
    - Multiple inheritance:
        - multiple parent classes and one child class
        - not supported in Java, reason: ambiguity
        - Example: A and B are parent classes, C is a child class of A and B
    - Hybrid inheritance:
        - combination of multiple inheritance and multilevel inheritance
        - Example: A and B are parent classes, C is a child class of A and B, D is a child class of C

- Rules:
    - The access modifier of the parent class should be same or less restrictive than the access modifier of the child class.
    - The return type of the methods can be same or different, but the parameters should be different.
    - The parameters can be different in terms of:
        - Number of parameters
        - Data type of parameters
        - Sequence of data type of parameters
    - The access modifier can be same or different.
    - The method name should be same.

Method Overriding

- Method overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes.
- It is used to add more to the behavior of methods
- Example:

```
class Vehicle {
    String color;
    int maxSpeed;
    public void display() {
        System.out.println("Color: " + color);
        System.out.println("Max speed: " + maxSpeed);
    }
}
class Car extends Vehicle {
    int numGears;
    boolean isConvertible;
    public void display() {
        super.display();
        System.out.println("Number of gears: " + numGears);
        System.out.println("Is convertible: " + isConvertible);
    }
}
```

- Rules:
  - The return type of the methods can be same or different, but the parameters should be different.
  - The parameters can be different in terms of:
    - Number of parameters
    - Data type of parameters
    - Sequence of data type of parameters
  - The access modifier can be same or different.
  - The method name should be same.
- `super` refers to the immeadiate parent class of the class

21/09/2023

Abstract classes

- An abstract class is a class that is declared with abstract keyword
- It can have abstract and non-abstract methods
- It cannot be instantiated
- It can have constructors and static methods
- It can have final methods which will force the subclass not to change the body of the method
- It can have final variables
- Syntax

```
abstract class ClassName {
    // body
}
```

- Example:

```
abstract class Vehicle {
    String color;
    int maxSpeed;
    public void display() {
        System.out.println("Color: " + color);
        System.out.println("Max speed: " + maxSpeed);
    }
    abstract public void accelerate();
    abstract public void brake();
}
class Car extends Vehicle {
    int numGears;
    boolean isConvertible;
    public void display() {
        super.display();
        System.out.println("Number of gears: " + numGears);
        System.out.println("Is convertible: " + isConvertible);
```

```
    }
    public void accelerate() {
        System.out.println("Car is accelerating");
    }
    public void brake() {
        System.out.println("Car is braking");
    }
}
```

- Can't create objects of abstract classes.
- Can't be instantiated, only inherited.
- Rules:
  - The access modifier of the parent class should be same or less restrictive than the access modifier of the child class.
  - The return type of the methods can be same or different, but the parameters should be different.
  - The parameters can be different in terms of:
    - Number of parameters
    - Data type of parameters
    - Sequence of data type of parameters
  - The access modifier can be same or different.
  - The method name should be same.
- Abstract Methods:
  - A method that is declared with abstract keyword and has no implementation is called an abstract method.
  - It is used to add more to the behavior of methods
  - Example:

```
abstract public void accelerate();
abstract public void brake();
```

  - Rules:
    - The return type of the methods can be same or different, but the parameters should be different.
    - The parameters can be different in terms of:
      - Number of parameters
      - Data type of parameters
      - Sequence of data type of parameters
    - The access modifier can be same or different.
    - The method name should be same.

Polymorphism

- Polymorphism is a feature that allows one interface to be used for a general class of actions.

- It is used to add more to the behavior of methods
- It is used to reuse the code
- types:
    - runtime - method overriding
    - compile time - method overloading

Encapsulation:

- Encapsulation is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.
- It is used to add more to the behavior of methods
- It is used to reuse the code
- It is used to achieve data hiding
- Example:

```
class Student {
    private int rollNo;
    private String name;
    private String address;
    public void setRollNo(int rollNo) {
        this.rollNo = rollNo;
    }
    public int getRollNo() {
        return rollNo;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    public String getAddress() {
        return address;
    }
}
```

Explanation of the example:

- `private` access modifier is used to achieve data hiding
- `this` keyword is used to refer to the current object
- `set` methods are used to set the values of the variables
- `get` methods are used to get the values of the variables
- `public` access modifier is used to make the methods accessible outside the class

Interface

- An interface is a collection of abstract methods
- It is a blueprint of a class.
- Three main reasons to use interfaces:
  - It is used to achieve abstraction
  - It is used to achieve multiple inheritance
  - It is used to achieve loose coupling
- Store only th function signatures and not the function definitions
  Coupling:
- Coupling is the degree of interdependence between software modules
- It is used to achieve loose coupling

Syntax:

```
interface InterfaceName {
    // body
}
```

Example:

```
interface Vehicle {
    public void accelerate();
    public void brake();
}

class Car implements Vehicle {
    int numGears;
    boolean isConvertible;
    public void accelerate() {
        System.out.println("Car is accelerating");
    }
    public void brake() {
        System.out.println("Car is braking");
    }
}
```

- Class *extends* class
- Interface *extends* interface
- Class *implements* interface

Example with a class implementing multiple interfaces:

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

```
interface Interface2 {
    public void method2();
}

class Class1 implements Interface1, Interface2 {
    public void method1() {
        System.out.println("Method 1");
    }
    public void method2() {
        System.out.println("Method 2");
    }
}
```

This overcomes the ambiguity problem of multiple inheritance.

Difference between interface and class tabulated:

Interface	**Class**
It is used to achieve abstraction	It is used to achieve encapsulation
It is used to achieve multiple inheritance	It is used to achieve single inheritance
It is used to achieve loose coupling	It is used to achieve tight coupling
It can have only abstract methods	It can have abstract and non-abstract methods
It can have only final variables	It can have final and non-final variables
It can have only static methods	It can have static and non-static methods
It can have only default methods	It can have default and non-default methods
It can have only public methods	It can have public, private, protected and default methods
It can have only public variables	It can have public, private, protected and default variables
It can have only public nested classes	It can have public, private, protected and default nested classes
It can have only public nested interfaces	It can have public, private, protected and default nested interfaces
It can have only public static nested classes	It can have public, private, protected and default static nested classes
It can have only public static nested interfaces	It can have public, private, protected and default static nested interfaces
It can have only public static final variables	It can have public, private, protected and default static final variables

Interface	Class
It can have only public static final nested classes	It can have public, private, protected and default static final nested classes

Disadvantages of interfaces:

- It cannot be instantiated
- It cannot have constructors
- It cannot have instance variables
- It cannot have instance methods
- It is slower in execution. It requires extra indirection to find the corresponding method in the actual class.

16/10/23:

Packages

todo

17/10/23:

Access Modifiers:

- used to set the assessiblity of classes, constructors, methods and properties.
- controls the visibilty
  Types:

1. Private: accessible only within the class
2. Default: accessible within the package only
3. Protected: accessible within the package and outside the package but through inheritance only
4. Public: accessible everywhere

Access Modifier	Within class	Within package	Outside package by subclass only	Outside package
Private	Yes	No	No	No
Default	Yes	Yes	No	No
Protected	Yes	Yes	Yes	No
Public	Yes	Yes	Yes	Yes

Access Modifier	Class	Constructor	Method	Property	Interface
Private	Yes	No	Yes	Yes	No
Default	Yes	Yes	Yes	Yes	No
Protected	Yes	Yes	Yes	Yes	No

Access Modifier	Class	Constructor	Method	Property	Interface
Public	Yes	Yes	Yes	Yes	Yes

Public:

- keyword: public
- accessible everywhere
- can be accessed by any class
- can be accessed by any package
- can be accessed by any subclass
- can be accessed by any class in any package
- can be accessed by any subclass in any package
  Example:

```
public class Student {
    public int rollNo;
    public String name;
    public String address;
    public void display() {
        System.out.println("Roll no: " + rollNo);
        System.out.println("Name: " + name);
        System.out.println("Address: " + address);
    }
}
```

Private:

- keyword: private
- accessible only within the class
- can be accessed by the same class
- can't be accessed by any other class
  Example:

```
public class Student {
    private int rollNo;
    private String name;
    private String address;
    public void display() {
        System.out.println("Roll no: " + rollNo);
        System.out.println("Name: " + name);
        System.out.println("Address: " + address);
    }
}
```

Default:

- keyword: not required
- accessible within the package only
- can be accessed by the same class
- can be accessed by any other class in the same package
- can't be accessed by any other class outside the package
  Example: