Unit 5

Object Oriented Programming

Syllabus

- Classes and objects: concepts of classes and objects
- Declaring objects, assigning object reference variables
- methods, constructors, access control, garbage collection
- usage of static with data and methods
- usage of final with data
- overloading methods and constructors.
- parameter passing - call by value, recursion
- nested classes.

Class:

- A blueprint or a template for creating objects. It defines the structure (**data/attributes**) and **behavior** (methods) that all objects of that class will have.

- Think of it as a cookie cutter. It defines the shape, but it's not the cookie itself
- **Object**: An instance of a class. It is a concrete, living entity created from the class blueprint. Each object has its own unique set of attribute values.

- Think of the actual cookies you bake using the cookie cutter. Each cookie is an object.

Analogy: The Blueprint and the House

- **Class = Architectural Blueprint**
- Defines the general design: number of bedrooms, bathrooms, square footage, materials to be used.
- It's a plan; it doesn't occupy physical space.
- **Object = Actual House**
- A specific, physical house built according to the blueprint.
- Each house (object) has its own unique characteristics: address, color, owner.

Defining a Class

- Syntax (General)
    - class ClassName { … }
- Class Members
    - Attributes (Data/Fields/Variables): Represent the state or characteristics of an object.
        - Example: string name;, int age;
    - Methods (Functions): Represent the behavior or actions an object can perform.
        - Example: void walk();, void speak();


```java
class Dog {
    // Attributes
    String name;
    String breed;
    int age;

    // Methods
    void bark() {
        // code to make the dog bark
    }

    void wagTail() {
        // code to make the dog wag its tail
    }
}
```

Declaring Object Reference Variables

- An object variable doesn't hold the object itself, but rather a **reference** to the object in memory.
- It's like having a remote control (the variable) that points to the TV (the object).
- **Syntax**:
  - ClassName objectReferenceVariable;

- **Example**:
  - Dog myDog;
  - At this point, myDog is null. It's a variable that can hold a Dog object reference, but it's not pointing to any object yet.

Creating an Object: The new Keyword

- The **new** keyword is used to instantiate (create an instance of) a class.
- It allocates memory for the new object and returns a reference to that memory location.
- Syntax:
    - new ClassName();

Assigning Object Reference Variables

- To make a reference variable point to a newly created object, you combine declaration and instantiation.
- **Syntax**:
  - ClassName objectName = new ClassName();
- **Example**:
  - Dog myDog = new Dog();
  - This statement does two things:
    - It creates a new Dog object in memory.
    - It assigns the memory address of this new object to the myDog variable.

Accessing Object Members

- Once an object is created, you can access its attributes and methods using the dot operator (.).
- Syntax:
- objectName.attributeName;
- objectName.methodName();
- Example (on slide):
- myDog.name = "Fido";
- myDog.age = 3;
- myDog.bark();

A Complete Code Example

```java
class Dog {
    String name;
    int age;

    void bark() {
        System.out.println(name + " says Woof!");
    }
}

public class Main {
    Run | Debug
    public static void main(String[] args) {
        // Declaring an object reference variable
        Dog myDog;

        // Instantiating a new Dog object and assigning the reference
        myDog = new Dog();

        // Setting the object's attributes
        myDog.name = "Buddy";
        myDog.age = 5;

        // Accessing the object's method
        myDog.bark();

        // Creating another object
        Dog anotherDog = new Dog();
        anotherDog.name = "Lucy";
        anotherDog.bark();
    }
```

Expected Output

- Buddy says Woof!
- Lucy says Woof!

Methods in Java

Understanding the concept, syntax, and practical use of methods

Introduction to Methods

- A method is a block of code that performs a specific task.
- Purpose: reusability, better organization, and modular programming.
- Every method belongs to a class and can be called multiple times.

```
public class Greeting {
    static void sayHello() {
        System.out.println("Hello!");
    }
    public static void main(String[] args) {
        sayHello(); // calling a method
    }
}
```

Syntax of a Method

```
// General form
<access-modifier> <return-type>
<methodName>(<parameters>) {
    // method body
}
```

Access Modifier: public, private, etc.
Return Type: int, void, String, etc.
Method Name: meaningful and camelCase.
Parameters: data your method needs.
Body: statements to perform the task.

Types of Methods

```
// Predefined (built-in) methods
System.out.println("Hello World");
double r = Math.sqrt(25);
```

```
// User-defined method
public void greet() {
    System.out.println("Hello, Java!");
}
```

Creating and Calling Methods

- ```
  public class Demo {
      void display() {
          System.out.println("Welcome to Java Methods!");
      }
      public static void main(String[] args) {
          Demo obj = new Demo(); // create object
          obj.display();         // call method
      }
  }
  ```
  ```

Method Parameters and Arguments

- Parameters: variables in the method signature.
- Arguments: actual values passed on call.

```
public class MathOps {
    public int multiply(int a, int b) {
        return a * b;
    }
    public static void main(String[] args) {
        MathOps m = new MathOps();
        int result = m.multiply(4, 5);
        System.out.println("Result: " + result);
    }
}
```

Return Type in Methods

- Methods may return a value, or use void if nothing is returned.
- Return type must match the value returned.

```
public class Student {
    public String getName() {
        return "Java Student";
    }
    public static void main(String[] args) {
        Student s = new Student();
        System.out.println(s.getName());
    }
}
```

Summary

- Methods group reusable logic and keep programs organized.
- Syntax: <access-modifier> <return-type> <name>(parameters) { body }
- Types: Predefined (library) and User-defined (your own).
- Practice writing small, focused methods with clear names.

Constructors in Java

Understanding the concept, types, and use of constructors

Introduction to Constructors

- A constructor is a special method used to initialize objects.
- Same name as the class.
- Called automatically when an object is created.
- No return type (not even void).
- Analogy: Setting up a new phone when you first get it.

```
public class Demo {
    Demo() {
        System.out.println("Object Created");
    }
    public static void main(String[] args) {
        Demo obj = new Demo();
    }
}
```

Syntax of a Constructor

- Name must match the class name.
- No return type.
- Can have parameters.
- Analogy: Filling out a 'Welcome Form' at a gym.

```
// General form
ClassName(parameters) {
    // initialization code
}
```

Types of Constructors

```
// Default Constructor
public class Student {
    String name;
    Student() {
        name = "Unknown";
    }
}
```

```
// Parameterized Constructor
public class Student {
    String name;
    Student(String n) {
        name = n;
    }
}
```

Constructor Overloading

- Multiple constructors with different parameters.
- Allows creating objects in different ways.
- Analogy: Same car model in basic, sports, and luxury versions.

```
public class Box {
    int length, width;
    Box() {
        length = width = 0;
    }
    Box(int l, int w) {
        length = l;
        width = w;
    }
}
```

this() Constructor Call

- Calls one constructor from another.
- Avoids code repetition.
- Analogy: Meal combo order calling single-item orders.

```
public class Car {
    String model;
    Car() {
        this("Unknown");
    }
    Car(String m) {
        model = m;
    }
}
```

Difference Between Constructor and Method

Constructor	Method
Same name as class	Any valid name
No return type	Must have a return type
Called automatically on object creation	Called manually
Used for initialization	Used for functionality
Analogy: Name at birth	**Analogy:** Task later in life

Summary

- Constructors run when objects are created.
- Two types: Default and Parameterized.
- Can be overloaded for flexibility.
- this() can call another constructor.
- Constructors are like the birth certificate of an object.

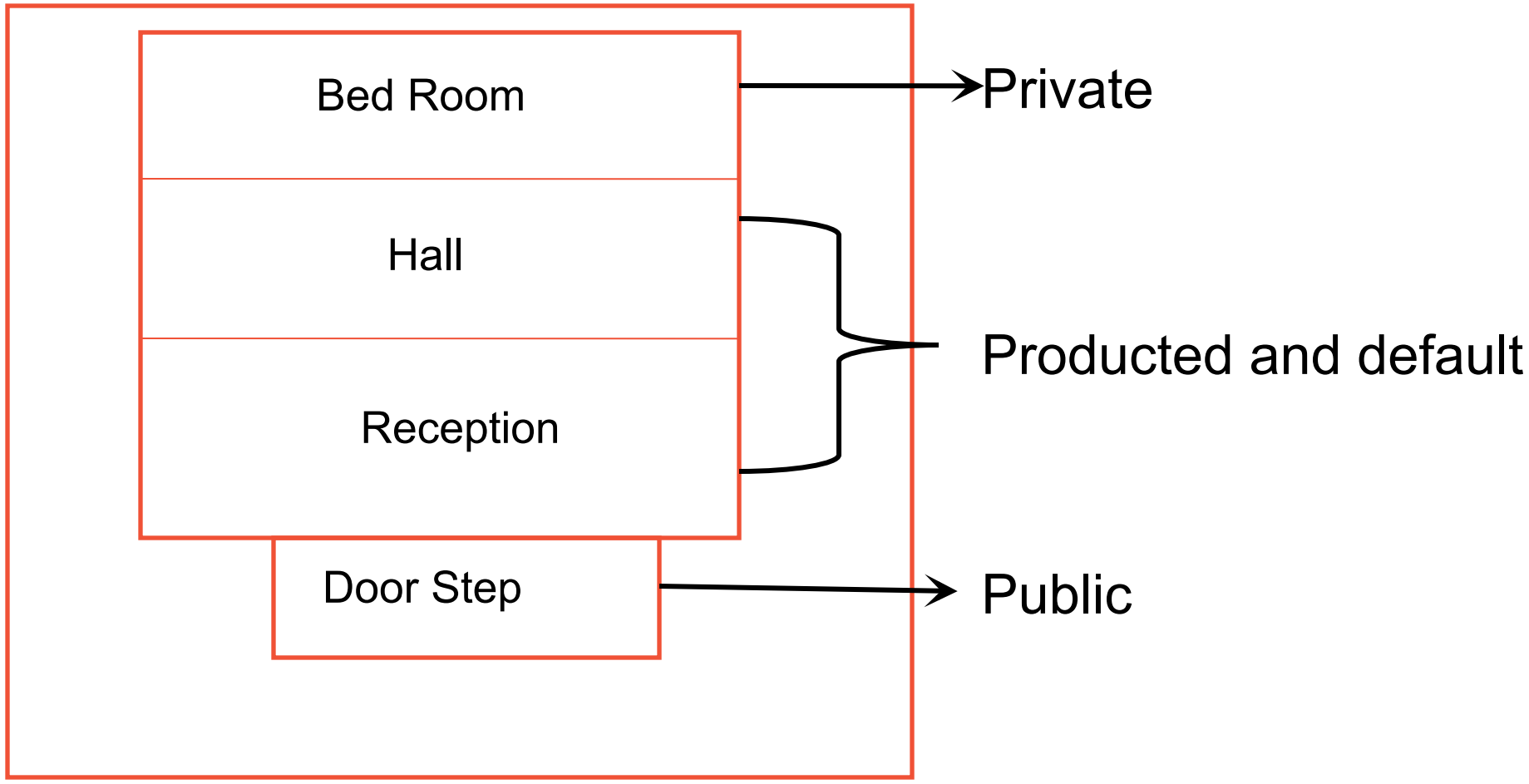
Access Modifiers

Helps to restrict the scope of a class, constructor , variable , method or data member.

Types

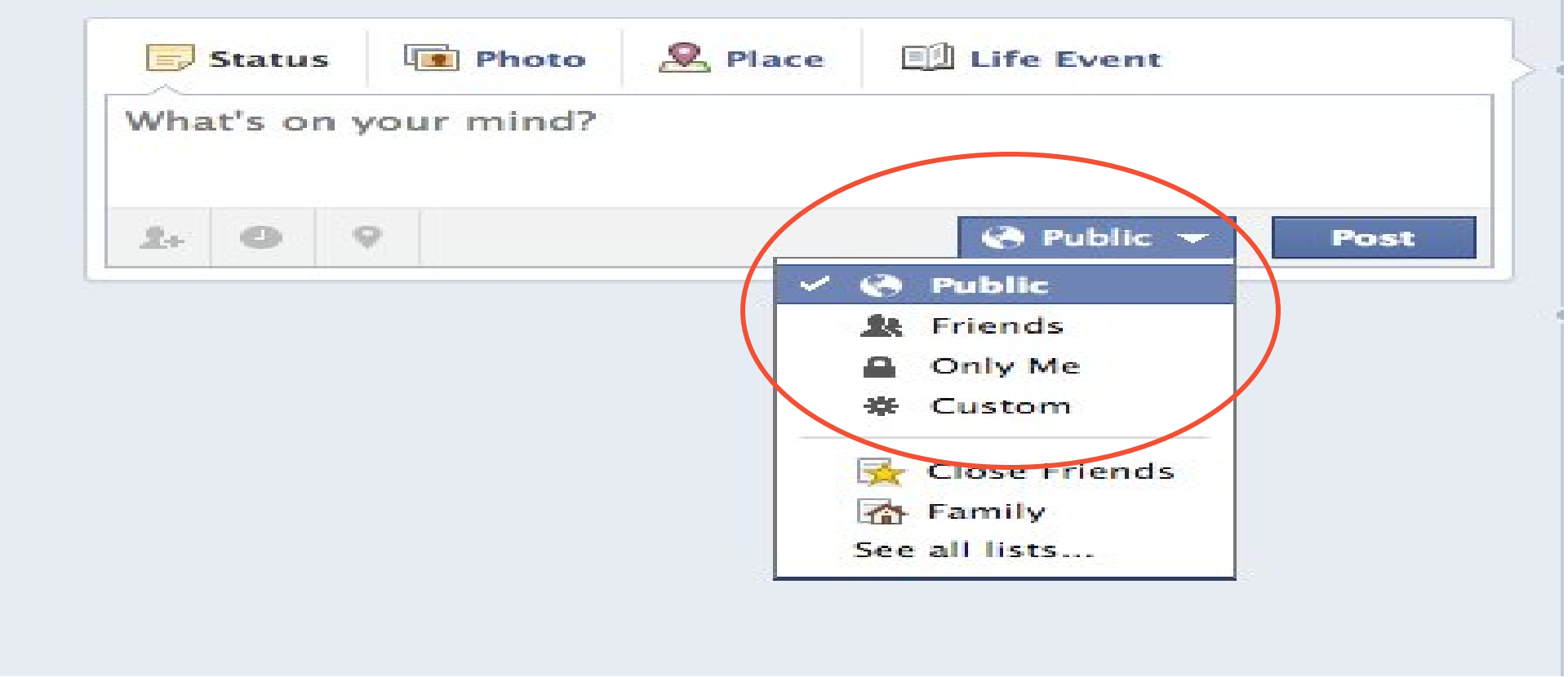
- Public
- Private
- Protected
- Default

Real time example



Real time application

Public


Status
Photo
Place
Life Event
What's on your mind?
Public
Post
Public
Friends
Only Me
Custom
Close Friends
Family
See all lists...

Public

- The public access modifier is specified using the keyword **public.**
- The members, methods and classes that are declared public can be accessed from anywhere.

```
1 //Predict the output
2
3   package one;
4   public class A
5  {
6    public void display()
7        {
8            System.out.println("Java");
9        }
10 }
11
12  import one.*;
13  class B
14 {
15    public static void main(String args[])
16       {
17           A obj = new A;
18           obj.display();
19       }
20 }
21
22
```

```
1   //Creating Packing countdigits
2
3   package countdigits;
4   public class First
5   {
6      public void no_of_digits()
7          {
8              int count = 1;
9              while(num>0){
10                 num=num/10;
11                 count++;
12             }
13             System.out.print(count);
14         }
15  }
16
17
18
19
20
21
22
```

```
1   //Predict the output
2
3   import countdigits.*;
4   class Second
5   {
6       public static void main(String args[])
7         {
8             First obj = new First();
9             obj.no_of_digits(456);
10        }
11  }
12
13
14
15
16
17
18
19
20
21
22
```

Private


Status
Photo
Place
Life Event
What's on your mind?
Public
Post
Public
Friends
Only Me
Custom
Close Friends
Family
See all lists...

Private

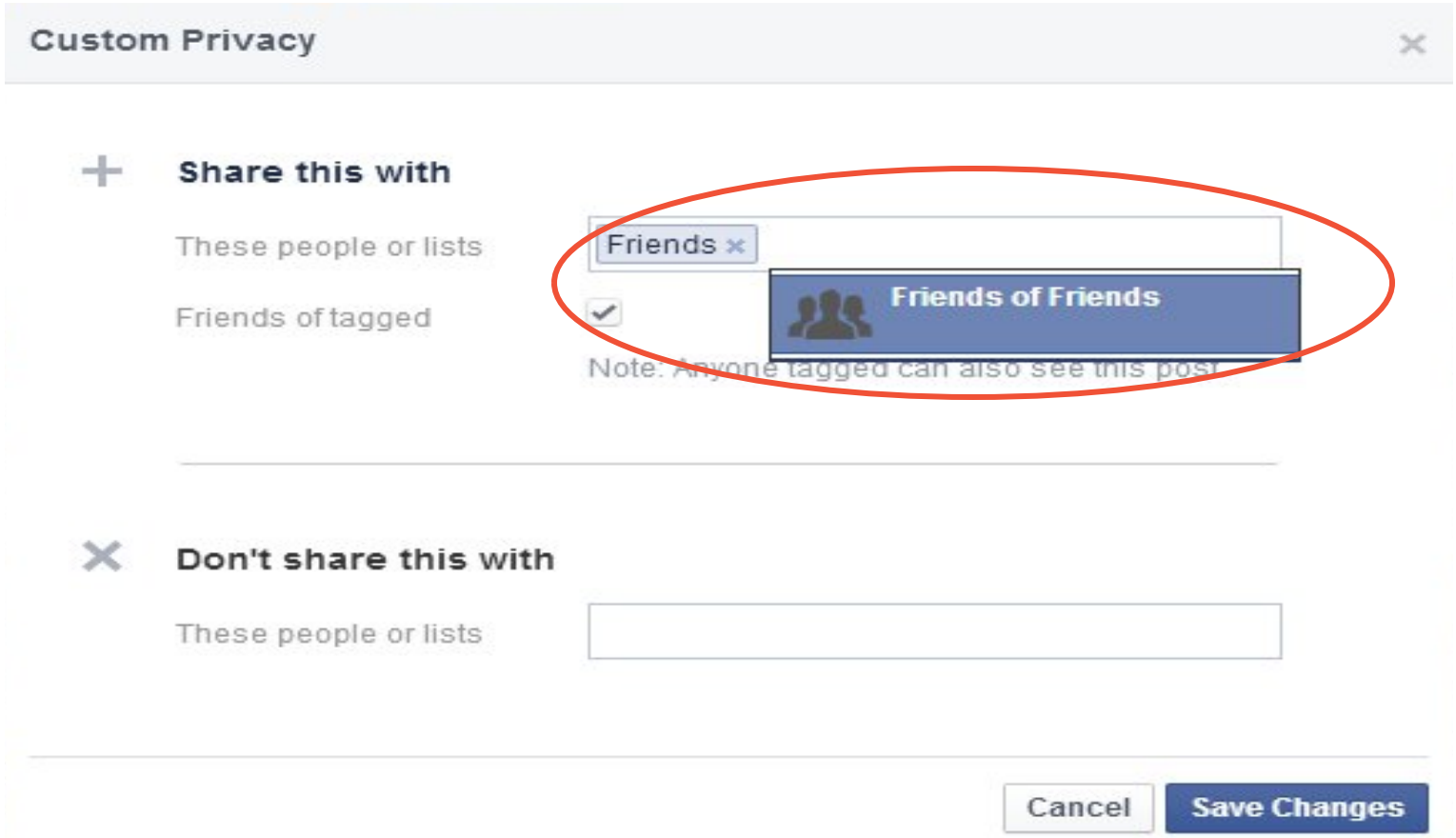
- The private access modifier is specified using the keyword **private.**
- The methods or data members declared as private are accessible only **within the class** in which they are declared.

```
1  //Predict the output
2
3  class A{
4     private double num = 100;
5     private int add(int a,int b){
6        return a+a;
7     }
8  }
9  public class Example{
10    public static void main(String args[]){
11       A obj = new A();
12       System.out.println(obj.num);
13       System.out.println(obj.add(10,20));
14    }
15 }
16
17
18
19
20
21
22
```

```
1  //Predict the output
2
3  public class PersonalData {
4      private String name;
5      public String getName() {
6          return this.name;
7      }
8      public void setName(String name) {
9          this.format = name;
10     }
11 }
12 public class UnknownPerson {
13     Public static void main(String[] main){
14         PersonalData d = new PersonalData();
15         d.setName("Mansoor");
16         System.out.println(d.getName());
17     }
18 }
19
20
21
22
```


```
1   //Predict the output
2
3    class Data
4   {
5       private String name;
6       public String getName(){
7           return this.name;
8       }
9       public void setName(String name)
10      {
11          this.name = name;
12      }
13  }
14  public class Main
15  {
16      public static void main(String[] main)
17      {
18          Data d = new Data();
19          d.setName("Focus");
20          System.out.println(d.getName());
21      }
22  }
```

Protected


Custom Privacy
Share this with
These people or lists
Friends
Friends of tagged
Friends of Friends
Note: Anyone tagged can also see this post
Don't share this with
These people or lists
Cancel
Save Changes

Protected

- The protected access modifier is specified using the keyword **protected.**
- It can be accessed within the same package classes and sub-class of any other packages.


```
1  //Predict the output
2
3  package package1;
4  public class Product {
5        protected int multiplication(int a, int b){
6        return a*b;
7     }
8  }
9  package package2;
10 import package1.*;
11 class Test extends Product{
12    public static void main(String args[]){
13       Test obj = new Test();
14       System.out.println(obj.multiplication(11,10));
15    }
16 }
17
18
19
20
21
22
```

Default

- The data members, class or methods which are not declared using any access modifiers.
- It is accessible only within the same package.

```
1   //Predict the output
2
3   class Logger
4   {
5       void message()
6       {
7           System.out.println("This is a default");
8       }
9   }
10
11
12
13
14
15
16
17
18
19
20
21
22
```

```
1  //Predict the output
2
3  package package1;
4  public class Product {
5        int multiplication(int a, int b){
6        return a*b;
7     }
8  }
9  package package2;
10 import package1.*;
11 class Test {
12    public static void main(String args[]){
13       Product obj = new Product();
14       System.out.println(obj.multiplication(11,10));
15    }
16 }
17
18
19
20
21
22
```

Summary

Access Modifiers	Visibility
Public	Visible to All classes.
Private	Visible with in the class. It is not accessible outside the class.
Protected	Visible to classes with in the package and the subclasses of other package.
No Access Modifier (Default)	Visible to the classes with the package

Garbage Collection in Java

Automatic memory management in Java programs

Introduction to Garbage Collection

How Garbage Collection Works


```
public class Test {
    public static void main(String[] args) {
        Test t = new Test();
        t = null; // Now eligible for GC
    }
}
```

Eligibility for Garbage Collection

- An object is eligible if:
- 1. All references are null.
- 2. It is no longer reachable from any active thread or static reference.
- Analogy: Forgetting someone's phone number — you can't contact them.

```
Test t1 = new Test();
Test t2 = new Test();
t1 = t2; // old object referenced by t1 is eligible for GC
```

Requesting Garbage Collection

- JVM decides when to run GC, but you can request it.
- Methods: System.gc() and Runtime.getRuntime().gc()
- Note: Requesting does not guarantee immediate execution.

```
public class GCExample {
    public static void main(String[] args) {
        System.gc();
        // or
        Runtime.getRuntime().gc();
    }
}
```

finalize() Method

- Called by GC before removing an object from memory.
- Used to release resources (rarely used in modern Java).
- Analogy: Saying goodbye before moving out of a house.

```
protected void finalize() {
    System.out.println("Object is garbage collected");
}
```

Advantages of Garbage Collection

- Automates memory management.
- Reduces memory leaks.
- Frees developer from manual cleanup.
- Analogy: Automatic dishwasher cleaning without scrubbing.

Summary

- GC cleans unused objects automatically.
- Objects are eligible when no references remain.
- You can request GC, but JVM decides when it runs.
- finalize() is called before object removal.

Overloading Methods and Constructors in Java

Enhancing flexibility by defining multiple versions of the same method or constructor

Introduction to Overloading

Method Overloading

- Same method name, different parameter lists.
- Improves code readability and flexibility.
- Analogy: Like a vending machine serving coffee in small or large size depending on choice.

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

Rules for Method Overloading

- Must change:
- - Number of parameters, OR
- - Type of parameters, OR
- - Order of parameters
- Cannot overload just by changing return type.

Constructor Overloading

- Same constructor name (class name), different parameter lists.
- Helps create objects in different ways.
- Analogy: Like buying the same phone model with different storage and color options.

```
public class Box {
    int length, width;
    Box() {
        length = width = 0; }
    Box(int l, int w) {
        length = l;
        width = w;}
}
```

Combining Method and Constructor Overloading

- A class can have both overloaded methods and constructors.

```
public class Car {
    String model;
    Car() { model = "Unknown"; }
    Car(String m) { model = m; }
void start() { System.out.println("Starting..."); }
    void start(String keyType) {
        System.out.println("Starting with " + keyType);
    }
}
```

Benefits of Overloading

- Improves code readability.
- Makes code more flexible and reusable.
- Allows multiple ways to perform a similar operation.
- Analogy: Like an elevator with multiple buttons to reach the same floor.

Summary

- Overloading applies to both methods and constructors.
- Differentiated by parameter list, not return type alone.
- Enhances flexibility in object creation and method usage.
- Analogy: Multiple doors to the same room — choose the one that fits best.

Nested Classes

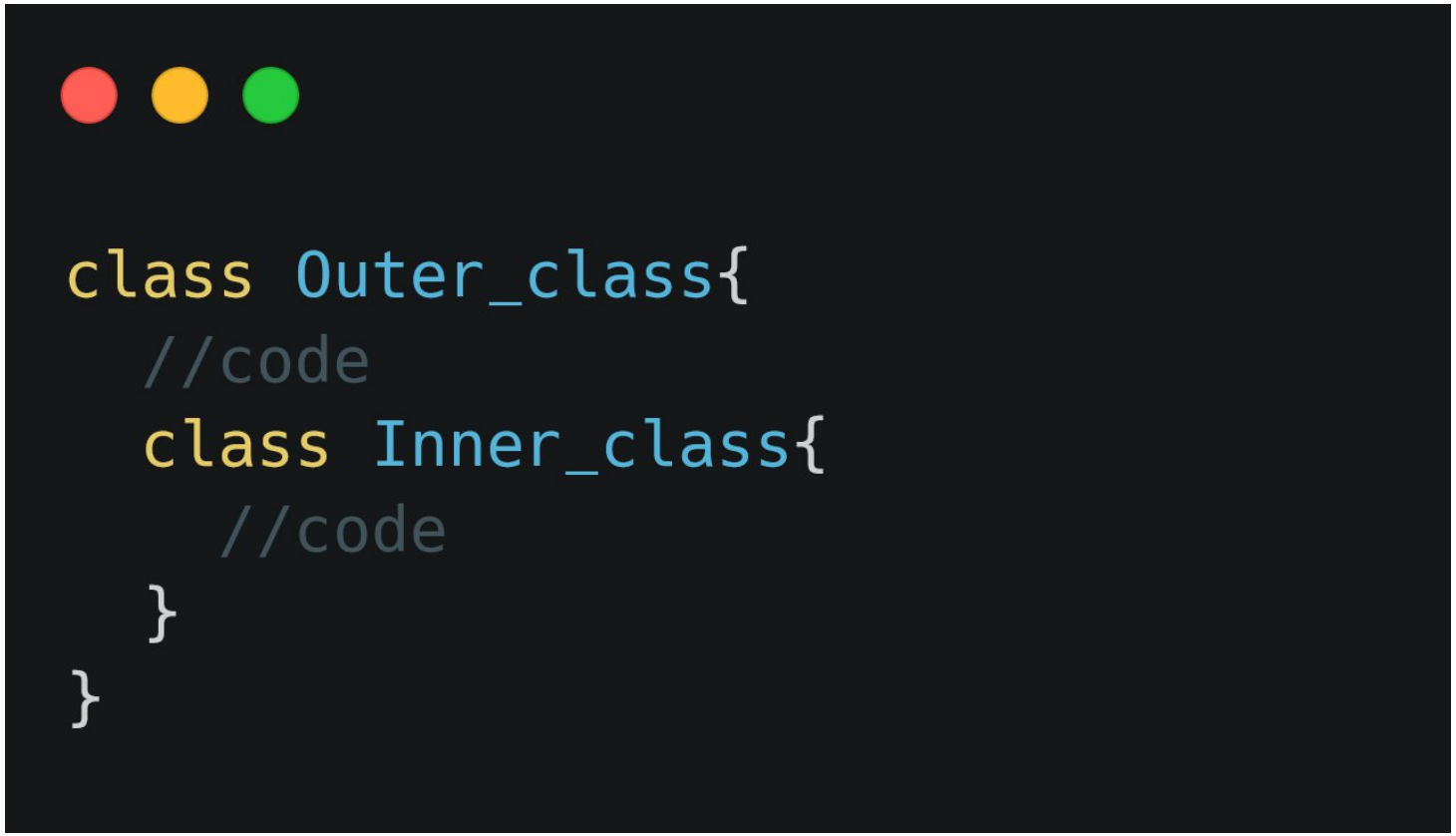
Introduction

- A nested class is a class defined within another class in Java.
- It helps logically group classes that are only used in one place, increasing encapsulation and readability.

Benefits:

- Logical grouping of classes
- Better encapsulation (can access outer class members)
- Code readability and maintainability

Syntax



```
class Outer_class{
  //code
  class Inner_class{
    //code
  }
}
```

Types

- **Static Nested Class** – Declared with the static keyword.
- **Non-static Inner Class** – Associated with an instance of the outer class.
  - **Local Inner Class** – Defined within a block (method, constructor, or block).
  - **Anonymous Inner Class** – Declared without a name, used for short-term usage.
  - **Member Inner Class** – declared inside the class but outside methods.

Static Nested Classes

- A nested class declared with static keyword.
- Can access **only static** members of the outer class.
- Does not require an object of the outer class to be instantiated.

```
//Psuedo code
class Car {
   static class Engine {
       void start() {
print("Engine started!"); }
   }
}


Car.Engine e = new Car.Engine();
e.start();
```

Non-Static Nested Classes

- Associated with an outer class object.
- Can access **all members** of the outer class (static, non-static, private).

- **Types**
  - **Local Inner Class –** Defined within a block (method, constructor, or block).
  - **Anonymous Inner Class –** Declared without a name, used for short-term usage.
  - **Member Inner Class –** Declared inside the class but outside methods.

Local Inner Class

- Defined within a block (method, constructor, or block)
- Generally, it is defined within a method.

```
class School {
    void conductExam() {
        // Local Inner Class (exists only inside this method)
        class Result {
            void showGrade(int marks) {
                if (marks >= 50) print("Pass");
                else print("Fail");
            }
        }

        // Using the local inner class
        Result r = new Result();
        r.showGrade(65);
    }
}

// Usage
School s = new School();
s.conductExam();
```

Member Inner Class

- Defined within the class and outside of

```
class Bank {
    private int balance = 1000;

    // Member Inner Class
    class Transaction {
        void withdraw(int amount) {
            balance -= amount;
            print("Withdrawn: " + amount);
            print("Remaining balance: " + balance);
        }
    }
}

// Usage
Bank b = new Bank();
Bank.Transaction t = b.new Transaction();
t.withdraw(200);
```

Anonymous Inner Class

- No explicit name
- Used for one-time implementation

```java
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Main {
    public static void main(String[] args) {
        Animal dog = new Animal() {//Anonymous Inner Class
            void sound() {
                System.out.println("Dog barks!");
            }
        };

        dog.sound();  // Output: Dog barks!
    }
}
```

Compression

Parameter	Member Inner Class	Local Inner Class	Anonymous Inner Class	Static Nested Class
Definition	Defined inside another class, but outside any method.	Defined inside a method, block, or constructor.	Nameless class declared and instantiated at the same time.	Declared as static inside another class.
Instantiation	Outer.Inner obj = outerObj.new Inner();	Created inside the method where defined.	Interface obj = new Interface() { ... };	Outer.StaticInner obj = new Outer.StaticInner();
Access to Outer Class Members	✔ Can access all (including private).	✔ Can access outer class + final/effectively final local variables.	✔ Can access outer class + final/effectively final local variables.	✖ Can only access static members directly.
Lifespan / Scope	Exists as long as the outer class object exists.	Exists only during method/block execution.	Exists as long as the object reference exists.	Exists independently of outer class object.
Example / Analogy	Bank → Transaction (always tied to a Bank).	School → Exam → Result (valid only during exam).	Button → onClickListener (one-time handler).	Car → Engine (Engine can exist without a Car object).