Core Java

Agenda

- Packages
- Access Specifies
- OOPS Explored

Introduction - Object Oriented Concepts

- Class
  - A blueprint that defines the attributes and methods
- Object
  - An instance of a Class
- Abstraction
  - Hide certain details and show only essential details
- Encapsulation
  - Binding data and methods together
- Inheritance
  - Inherit the features of the superclass
- Polymorphism
  - One name having many forms

What is Package?

- A package is a grouping of related types providing access protection and name space management
- Note that types refers to classes, interfaces, enumerations, and annotation types.
- Types are often referred to simply as classes and interfaces since enumerations and annotation types are special kinds of classes and interfaces, respectively, so types are often referred to simply as classes and interfaces.

Creating a Package

- To create a package, you choose a name for the package and put a **package** statement with that name at the top of every source file that contains the types (classes, interfaces, enumerations, and annotation types) that you want to include in the package
- If you do not use a package statement, your type ends up in an unnamed package
- Use an unnamed package only for small or temporary applications.

Example:

package com.altisource.examples;

package org.altisource.examples;

Importing and Using classes from other Packages

To use a public package member (classes and interfaces) from outside its package, you must do one of the following.

- Import the package member using import statement
- Import the member's entire package using import statement
- Refer to the member by its fully qualified name (without using import statement)

Example:

Importing a Class or a Package Importing a class

```
import java.util.Date;
```

Importing all classes in the java.util package

```
import java.util.*;
```

Using Classes of other packages via fully qualified path

```
public static void main(String[] args) {
    java.util.Date x = new java.util.Date();
}
```

Benefits of Packaging

- You and other programmers can easily determine that these classes and interfaces are related.
- You and other programmers know where to find classes and interfaces that can provide graphics related functions.
- The names of your classes and interfaces won't conflict with the names in other packages because the package creates a new namespace.
- You can allow classes within the package to have unrestricted access to one another yet still restrict access for types outside the package.

Access Specifiers

There are four access Specifiers:

- public
- protected
- default
- private

public:

- Specifies that class variables and methods are accessible to anyone, both inside and outside the class.
- Have global visibility and can be accessed by any other objects.

protected:

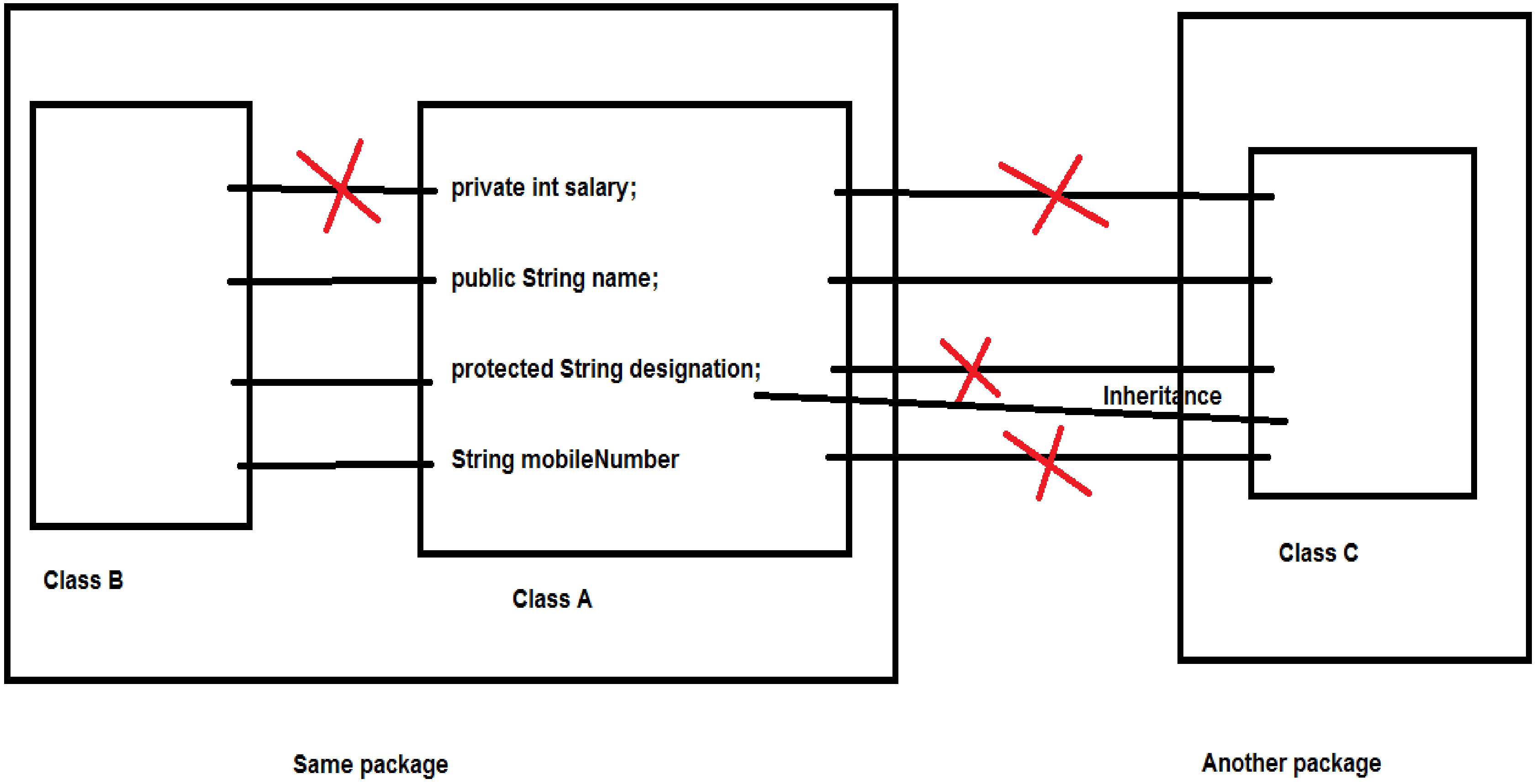
- Specifies that class members are accessible only to methods in that class and subclasses of that class
- Have visibility limited to subclasses

Default:

- Specifies that only classes in the same package can have access to a class’s variables and methods
- Class members with default access have a visibility limited to other classes within the same package
- There is no actual keyword for declaring the default access modifier

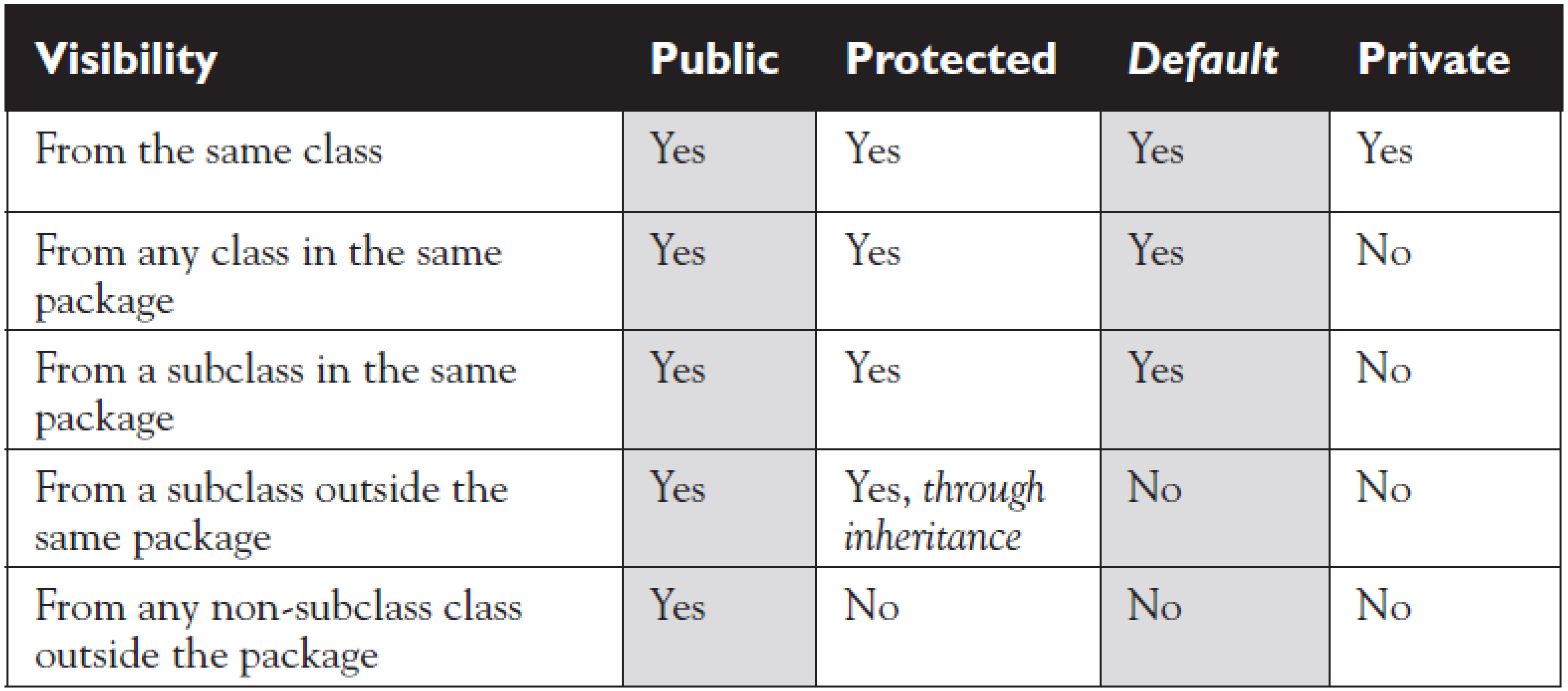
private:

- Is the most restrictive
- Specifies that class members are only accessible by the class they are defined in.
- No other class has access to private class members, even subclasses


private int salary;
public String name;
protected String designation;
String mobileNumber
Inheritance
Class B
Class A
Class C
Same package
Another package

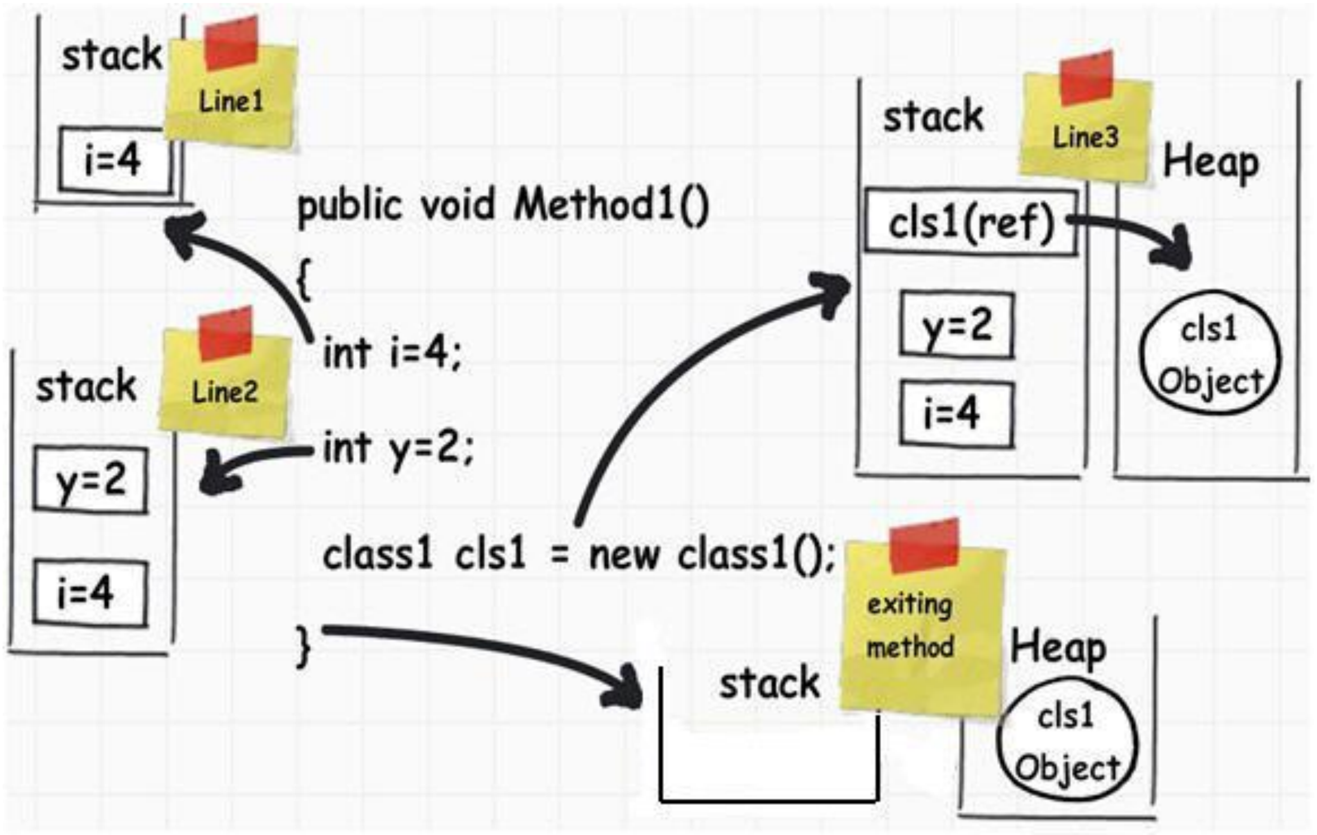
Protected members of class A are available to class B, but not in class C. But if class C is a sub class of class A , then the protected member of class A are available to class C.

So protected access specifier acts as public with respect to sub classes.



Visibility	**Public**	**Protected**	***Default***	**Private**
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, *through inheritance*	No	No
From any non-subclass class outside the package	Yes	No	No	No

Stack vs Heap



Java Modifiers

Modifier	Class	Class Variables	Methods	Method Variables
public	✓	✓	✓	
private		✓	✓	
protected		✓	✓	
default	✓	✓	✓	
final	✓	✓	✓	✓
abstract	✓		✓	
strictfp	✓		✓	
transient		✓		
synchronized			✓	
native			✓	
volatile		✓		
static	✓	✓	✓	

Modifiers – Class

- public
  - Class can be accessed from any other class present in any package
- default
  - Class can be accessed only from within the same package. Classes outside the package in which the class is defined cannot access this class
- final
  - This class cannot be sub-classed, one cannot extend this class
- abstract
  - Class cannot be instantiated, need to sub-classs/extend.
- strictfp
  - Conforms that all methods in the class will conform to IEEE standard rules for floating points

Modifiers – Class Attributes

- public
  - Attribute can be accessed from any other class present in any package
- private
  - Attribute can be accessed from only within the class
- protected
  - Attribute can be accessed from all classes in the same package and sub-classes.
- default
  - Attribute can be accessed only from within the same package.
- final
  - This value of the attribute cannot be changed, can assign only 1 value
- transient
  - The attribute value cannot be serialized
- volatile
  - Thread always reconciles its own copy of attribute with master.
- static
  - Only one value of the attribute per class

Modifiers – Methods

- public
  - Method can be accessed from any other class present in any package
- private
  - Method can be accessed from only within the class
- protected
  - Method can be accessed from all classes in the same package and sub-classes.
- default
  - Method can be accessed only from within the same package.
- final
  - The method cannot be overridden
- abstract
  - Only provides the method declaration
- strictfp
  - Method conforms to IEEE standard rules for floating points
- synchronized
  - Only one thread can access the method at a time
- native
  - Method is implemented in platform dependent language
- static
  - Cannot access only static members.

Constructors

- Creates instances for Classes
- Same name as Class name
- Can have any access modifier

```
Employee emp = new Employee()
```

```
public class Employee  {
    public int empid;
    public String name;
    public Employee() {
      System.out.println("Default constructor");
    }
    public Employee(int empid) {
      this.empid = empid;
    }
    public Employee(String name, int empid) {
      this.name = name;
      this.empid = empid;
    }
}
```

- Constructor does not have return type.
- Constructor cannot be overridden and can be over loaded.
- Default constructor is automatically generated by compiler if class does not have constructor.
- If explicit constructor is there in the class the default constructor is not generated.
- First statement should be a call to this() or super().

Overloaded Methods

Overloaded methods let you reuse the same method name in a class, but with different arguments (and optionally, a different return type).

Rules for overloading:

- Overloaded methods MUST change the argument list.
- Overloaded methods CAN change the return type.
- Overloaded methods CAN change the access modifier.
- Overloaded methods CAN declare new or broader checked exceptions.

Example:

public void changeSize(int size, String name, float pattern) { }

The following methods are legal overloads of the changeSize() method:

public void changeSize(int size, String name) { }

public int changeSize(int size, float pattern) { }

public void changeSize(float pattern, String name) throws IOException { }

Constructor Overloading:

```
public class ConstructorOverLoading {

public int radius;
public int length;
public int hight;
public ConstructorOverLoading(){
		System.out.println("Default Constructor..");
}

public ConstructorOverLoading(int radius){
		this.radius = radius;
}
public ConstructorOverLoading(int length, int hight){
		this.length = length;
		this.hight = hight;
}

}
```

this keyword:

- can be used to invoke a constructor of the same class.
- this avoids the naming ambiguity between local variables and instance variables.

Method Overloading:

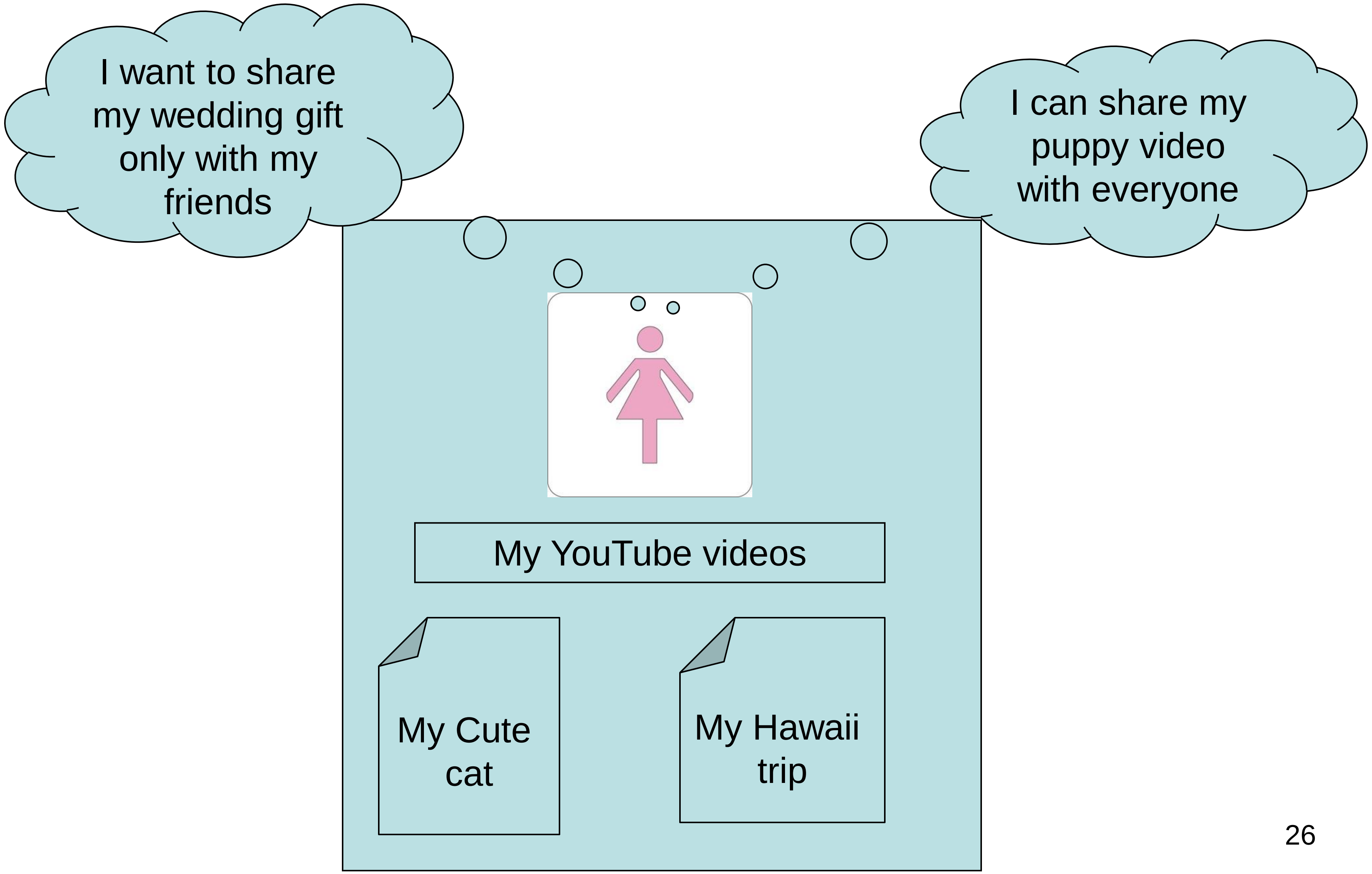
```
public class Area {

public double area(float radius) {
            System.out.println("Area of Circle..");
            return ((22 / 7) * (radius * radius));
    }

public double area(float length,float breth) {
            System.out.println("Area of rectangle..");
            return (length * breth);
   }

public double area(int side) {
            System.out.println("Area of Square..");
            return (side * side);
   }
}
```

OOPS Explored - Encapsulation


I want to share my wedding gift only with my friends
I can share my puppy video with everyone
My YouTube videos
My Cute cat
My Hawaii trip

OOPS Explored - Encapsulation

- **Encapsulation**
  Binding data and methods together

```
public class Employee
{
    private String empName;
    private int salary;

    public String getSalary(String role)
    {
        if(“Manager”.equals(role))  {
            return salary;
        }
    }

    public String setSalary(String role, int newSal)
    {
        if (“Admin”.equals(role))  {
            salary = newSal;
        }
    }
}
```

OOPS Explored - Abstraction

- **Abstraction**

  Hide certain details and show only essential details

```
public abstract class Shape
{
        String color;
         public abstract double getArea();
}
```

```
public interface Shape
{
        String static final String color = “BLACK”;
         public abstract double getArea();
}
```

- **Abstract class v/s interface**