OOP

ENCAPSULATION

INHERITANCE

Part 1

softserve

AGENDA

- Java OOPs Concepts
- Encapsulation
- Inheritance
- Abstract classes
- Composition
- Reference types

Java OOPs Concepts

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Java OOPs Concepts

Inheritance

When one object acquires all the properties and behaviors of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

When one task is performed by different ways i.e. known as polymorphism. For example: cat speaks meow, dog barks woof etc.

Abstraction

Hiding internal details and showing functionality is known as an abstraction. For example: phone call, we don't know the internal processing.

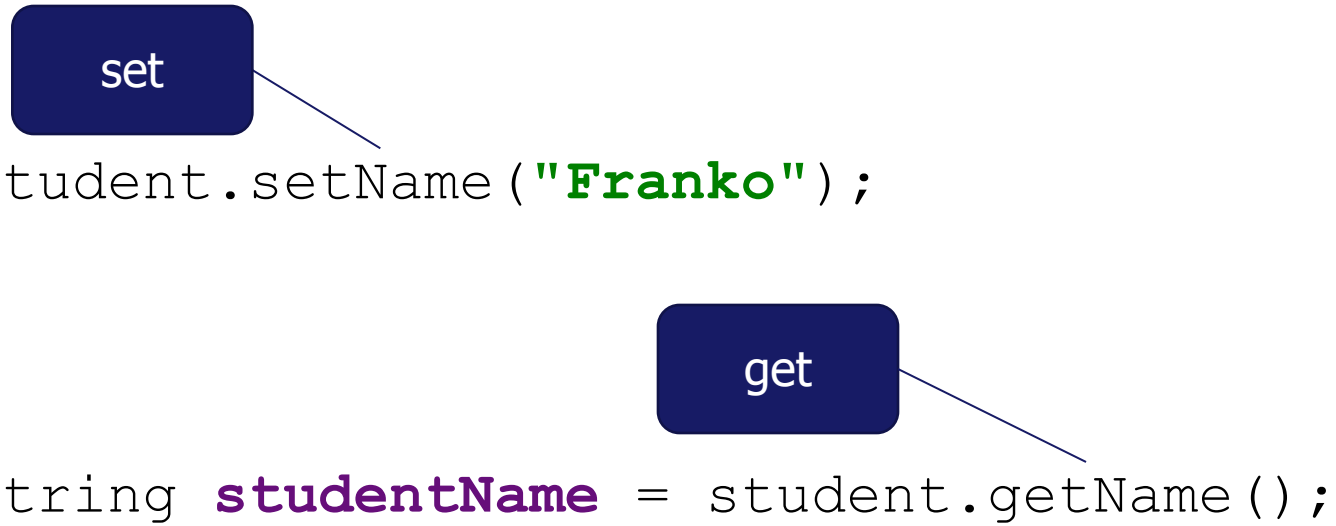
In java, we use abstract class and interface to achieve abstraction.

Encapsulation

- **Encapsulation** in Java is a mechanism of ***wrapping*** the data (variables) and code acting on the data (methods) together as a ***single unit***.
- Encapsulation is used to ***hide*** the ***values*** or ***state*** of a structured data object inside a class, preventing unauthorized parties direct access to them.
- Benefits of Encapsulation:
  - the fields of a class can be made ***read-only*** or ***write-only***;
  - a class can have ***total control*** over what is stored in its fields;
  - ***Isolation*** your public interface ***from change*** (allowing your public interface to stay constant while the implementation changes without affecting existing consumers).

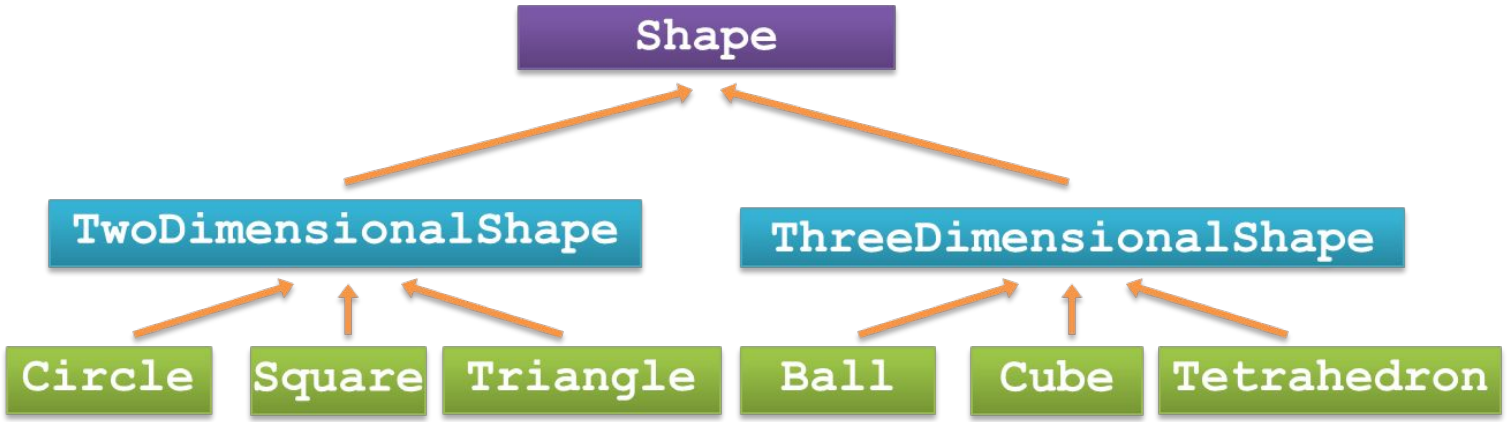
Getters and Setters

```
Student student = new Student();
```

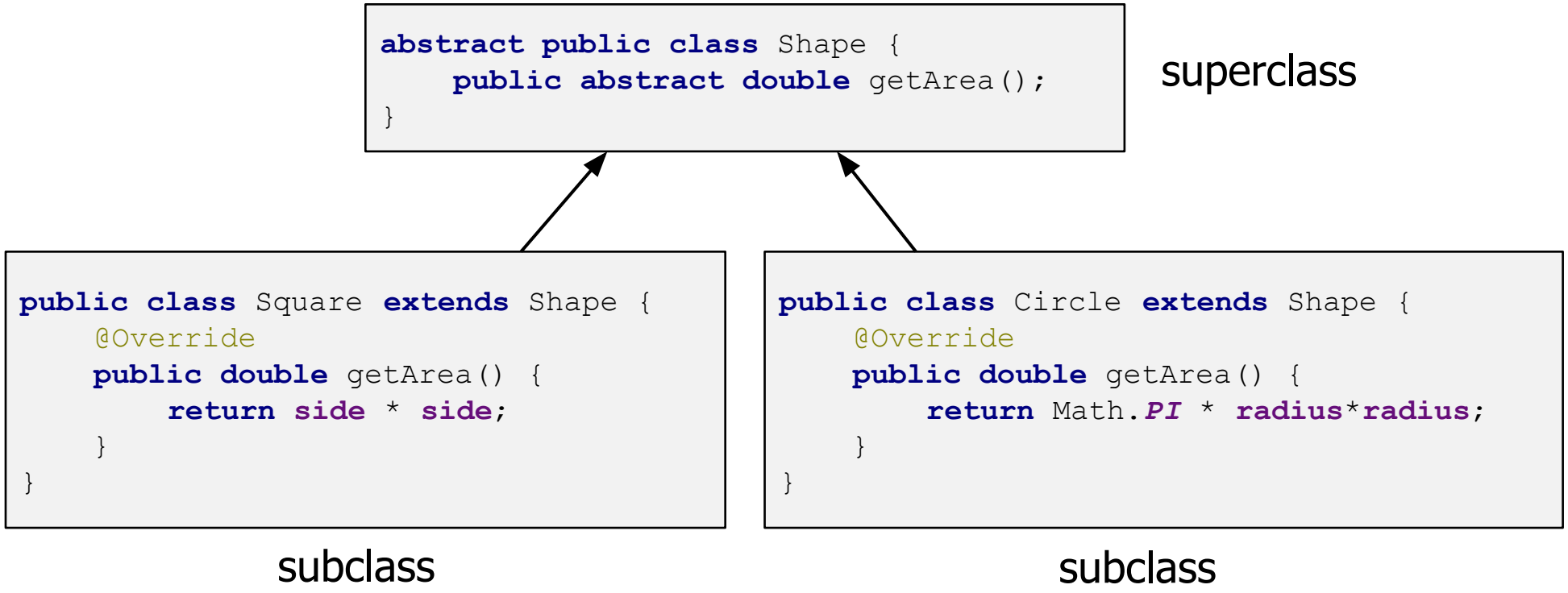


Inheritance

- ***Inheritance*** in Java is form of software ***reusability***:
  - new classes created from existing ones;
  - absorb attributes and behaviors and add in their own.
- Subclass inherits from superclass:
  - ***direct superclass*** – subclass explicitly inherits;
  - ***indirect superclass*** – subclass inherits from two or more levels up the class hierarchy.



Inheritance



Inheritance

- Example

```
public class Rectangle {
    public int width;
    public int height;

    public int getPerimeter() {
        return 2 * (width + height);
    }
}
```

- To ***inherit*** the ***properties*** and ***methods*** of a class you use the **`extends`** keyword.

```
public class Parallelogram extends Rectangle {
    public int angle;
}
```

Inheritance

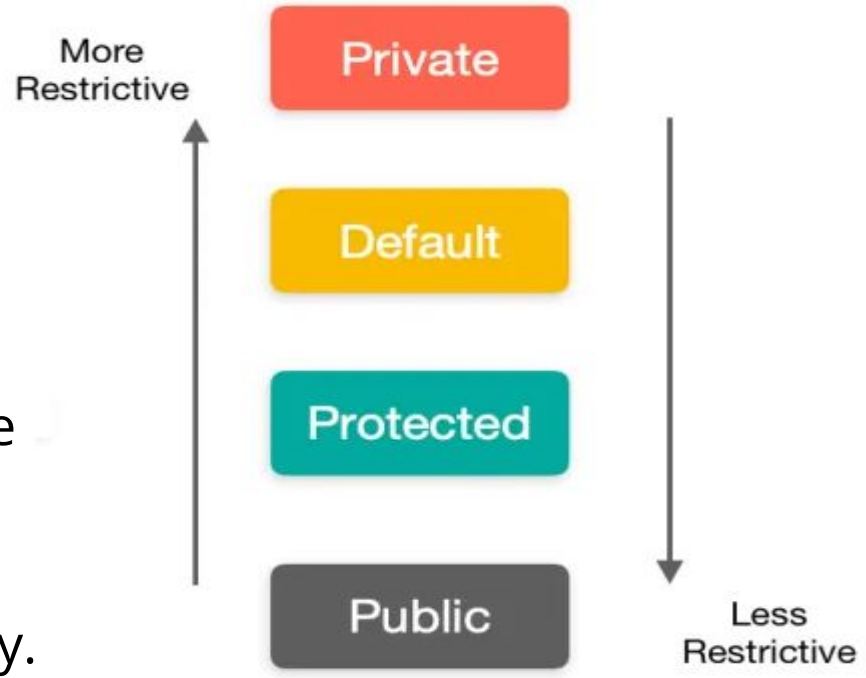
Example

```
public class Main {
    public static void main(String[] args) {

        Rectangle rectangle = new Rectangle();
        rectangle.width = 42;
        rectangle.height = 74;

        Parallelogram parallelogram = new Parallelogram();
        parallelogram.width = 42;   // inherit from Rectangle
        parallelogram.height = 74;  // inherit from Rectangle
        parallelogram.angle = 35;

        double p = parallelogram.getPerimeter(); // inherit from Rectangle
        System.out.println("Perimeter of parallelogram equals " + p);
    }
}
```

```
Perimeter of parallelogram equals 232.0
```

Access Modifiers

- Java provides a number of **Access Modifiers** to set access levels for classes, ***variables***, ***methods***, and ***constructors***.
- There are **4 types** of access levels:
  - **`public`** – visible to the everywhere
  - **`private`** – visible only in the same class
  - **default (package-private)** – visible within the package level
  - **`protected`** – within package and outside the package but need to use inheritance then only.

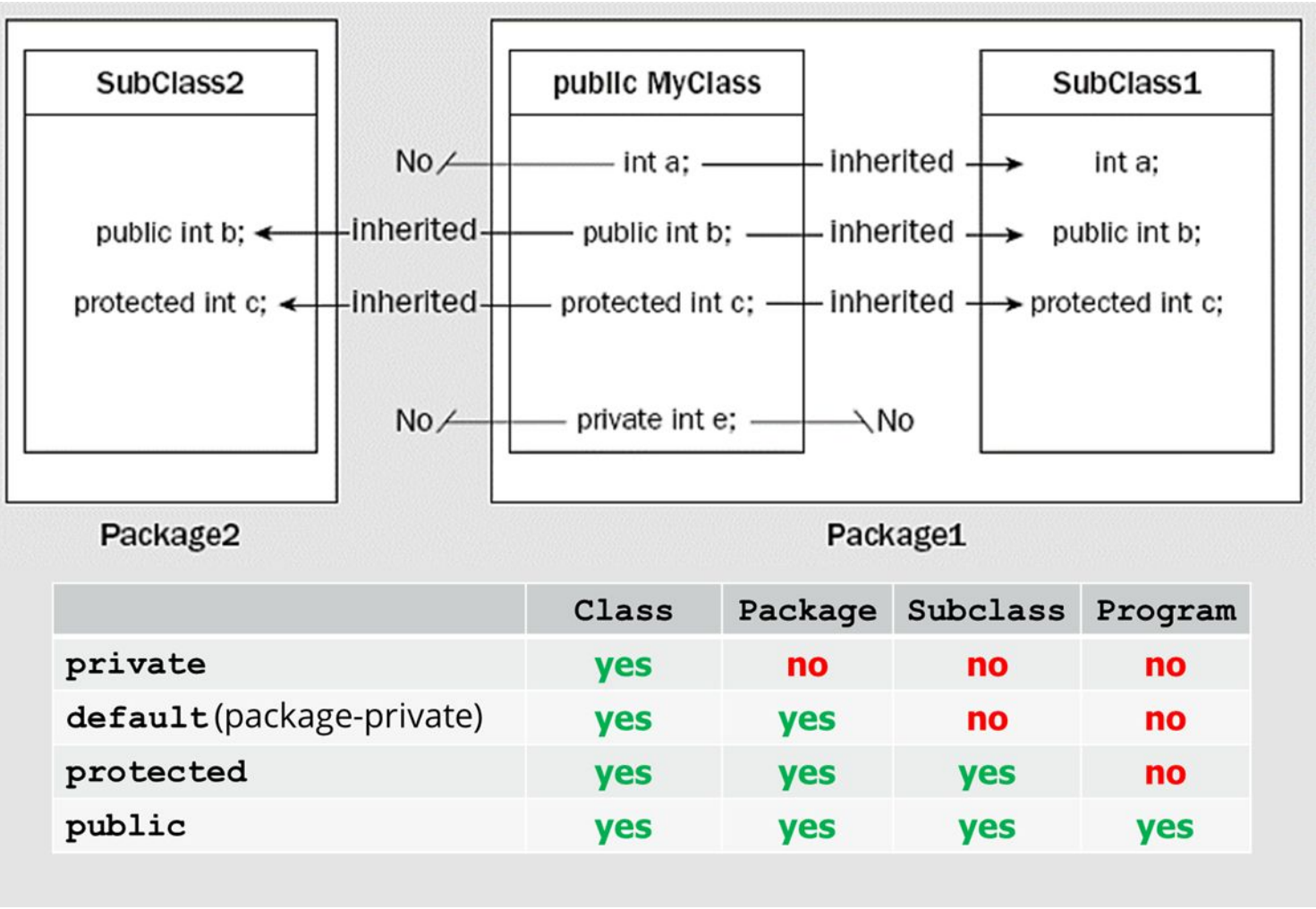


The `protected` Access Modifier

- **Variables**, **methods**, and **constructors**, which are declared `protected` in a **superclass** can be accessed only by the **subclasses** in ***other package*** or any ***class within the package***.
- The `protected` access modifier ***cannot be applied*** to **class** and **interfaces**.
- Example:

```
class Rectangle {
    protected int width;
    protected int height;
    // getters and setters
}

class Parallelogram extends Rectangle {
    private int angle;
    // getters and setters
    public int getArea() {
        return (int) (width * height * Math.sin(angle * Math.PI / 180));
    }
}
```

Access to Class Members



	Class	Package	Subclass	Program
`private`	yes	no	no	no
`default` (package-private)	yes	yes	no	no
`protected`	yes	yes	yes	no
`public`	yes	yes	yes	yes

softserve

Inheritance and Methods Overriding

- A subclass can ***modify behavior*** inherited from a parent class.
- A subclass can create a method with different functionality than the parent's method but with the ***same signature***.

```
class Rectangle {
    protected int width;
    protected int height;

    public int getPerimeter() { return 2 * (width + height); }

    public int getArea() { return width * height; }
}

class Parallelogram extends Rectangle {
    private int angle;

    public int getArea() {
        return (int) (width * height * Math.sin(angle * Math.PI / 180));
    }
}
```

The access modifier of an overriding or hiding method must provide at ***least as much*** access as the overridden or hidden method.

softserve

Abstract Classes

A class must be declared ***abstract*** when we need to forbid creating instances of this class.

Abstract class may have one or more ***abstract methods***.

A method is declared abstract when it has a method heading, but no body – which means that an abstract method has no implementation code inside curly braces like normal methods do.

- The derived class must provide a definition method;
- The derived class must be declared abstract itself.

A non abstract class is called a concrete class.

Abstract Classes

```
public abstract class Figure {
    /* because this is an abstract method the body will be blank */
    public abstract double getArea();
}

public class Circle extends Figure {
    private double radius;
    public Circle (double radius) { this.radius = radius; }
    public double getArea() { return (3.14 * (radius * 2)); }
}
```

Abstract Classes

```
public class Rectangle extends Figure {
  private double length, width;

  public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
  }

  public double getArea() { return length * width; }
}
```

Composition

- **Composition** is the design technique to implement **has-a** relationship in classes.
- Composition is achieved by using ***instance variables*** that ***refers*** to other objects.

```
class Point {
    private int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

```
class Circle {
    private Point point;
    private int radius;
    public Circle(Point point, int radius) {
        this.point = point;
        this.radius = radius;
    }
}
```

```
Circle circle = new Circle(new Point(74, 38), 26);
```

Inheritance vs. Composition

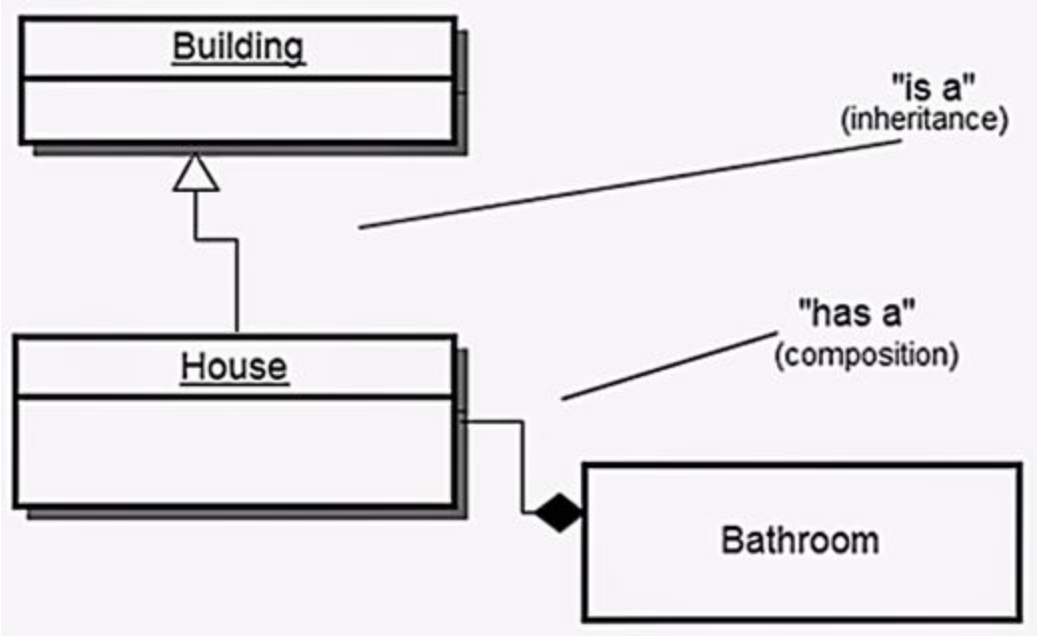
```
class Point {
    protected int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

```
class Circle extends Point {
    private int radius;
    public Circle(int x, int y, int radius) {
        super(x, y);
        this.radius = radius;
    }
}
```

```
Circle circle = new Circle(74, 38, 26);
```

Inheritance vs. Composition

- Kinds of **Relationships** between objects:
  - "**is a**" - object of subclass "is a" object of the superclass (**inheritance**).
  - "**has a**" – object "has a" object of another class as a member (**composition**);



Casting Objects

Assignment operator. What will be done ?

```
int num = 1;
double data = 1.0;
data = num;                 //  num = data; ???
```

Casting Objects

Assignment operator. What will be done ?

```
class Aclass { int field1 = 10; }

class Bclass extends Aclass { int field2 = 20;  }

Aclass a = new Aclass( );
Bclass b = new Bclass( );
a = b;                  //  b = a; ???
```

Casting Objects

- ***Upcasting*** is casting a ***subtype*** to a ***supertype***, upward to the inheritance tree.

```
Ball ball = new Ball(6.3, "MyBall");
ball.getVolume();
Shape shape = (Shape) ball;
shape.getVolume();
```

- ***Downcasting*** is casting a ***supertype*** to a ***subtype***, downward to the inheritance tree.

```
Shape shape = new Ball(6.3, "MyBall");
shape.getVolume();
Ball ball = (Ball) shape;
ball.getVolume();
```

Example

```java
package com.softserve.train;

public class Parent {

    int f( ) { return 1; }

    public int useF() {
        return f();
    }
}
```

```java
package com.softserve.train2;

import com.softserve.train.Parent;

public class Child extends Parent {

    int f() {
        return 2;
    }
}
```

Let's check it

```
package com.samples;

import com.softserve.train2.*;

public class OOPSamples {
    public static void main(String... args) {
        Child child = new Child();
        System.out.println(child.useF());
    }
}
```

Keyword **super**

- A constructor can call another constructor in its superclass using the keyword **super** and the parameters list.

```
public Rectangle(int w, int h) {
    width = w;
    height = h;
}
```

```
public Parallelogram(int w, int h, int a) {
    super(w, h);
    angle = a;
}
```

- The keyword **super** also used for access original superclass method.

```
public int getArea() {
    if (angle == 90) {
        return super.getArea();
    }
    return (int) (width * height * Math.sin(angle * Math.PI / 180));
}
```

Inheritance

```
public class Circle {
private double radius;

   // Constructors
   public Circle() { this.radius = 1.0; }
   public Circle(double radius) { this.radius = radius; }

   // Getters and Setters
   // Return the area of this Circle
   public double getArea() { return radius * radius * Math.PI; }
}
```

Inheritance

```
public class Cylinder extends Circle {

   private double height;

   // Constructors
   public Cylinder() {
      super();  // invoke superclass' constructor Circle()
      this.height = 1.0;
   }
```

Inheritance

```
public Cylinder(double height) {
   super();  // invoke superclass' constructor Circle()
   this.height = height;
}

public Cylinder(double height, double radius) {
   // invoke superclass' constructor Circle(radius)
   super(radius);
   this.height = height;
}
```

Inheritance

```
    // Getter and Setter
    // Return the volume of this Cylinder
    public double getVolume() {
        // Use Circle's getArea()
        return getArea() * height;
    }

    // Describe itself
    public String toString() { return "This is a Cylinder"; }
}
```

Inheritance

```
public class ClassA {
   public int i = 1;
   public void m1() { System.out.println("ClassA, metod m1, i = " + i); }
   public void m2() {   System.out.println("ClassA, metod m2, i = " + i);
}
   public void m3() {
      System.out.print("ClassA, metod m3, runnind m4():"); m4();  }
   public void m4() { System.out.println("ClassA, metod m4"); }
}
```

Inheritance

```
public class ClassB extends ClassA {
    public double i = 1.1;
    public void m1() { System.out.println("ClassB, metod m1, i= " + i); }
    public void m4() { System.out.println("ClassB, metod m4"); }
}
```

Automatically added default constructor.

Inheritance

```
public class AplAB {
   public static void main(String[] args) {

       System.out.println("The Start.");
       ClassA a = new ClassA();
       System.out.println("Test ClassA.");
       a.m1();
       a.m2();
       a.m3();
       a.m4();
```

Inheritance

```
ClassB b = new ClassB();
System.out.println("Test ClassB.");
b.m1();
b.m2();
b.m3();
b.m4();
```

```
ClassA b0 = new ClassB();
      System.out.println("Test_0 ClassB.");
      b0.m1();
      b0.m2();
      b0.m3();
      b0.m4();
    }  }
```

Practical tasks

1. Create abstract class *Car* with *model*, *maxSpeed* and *yearOfManufacture* properties and run() and stop() methods.

Develop classes Truck and Sedan which extend class Car.

In main method create array of Car’s objects. Add to this array some trucks and sedans and print info about it

Practical tasks

2. Create three classes:

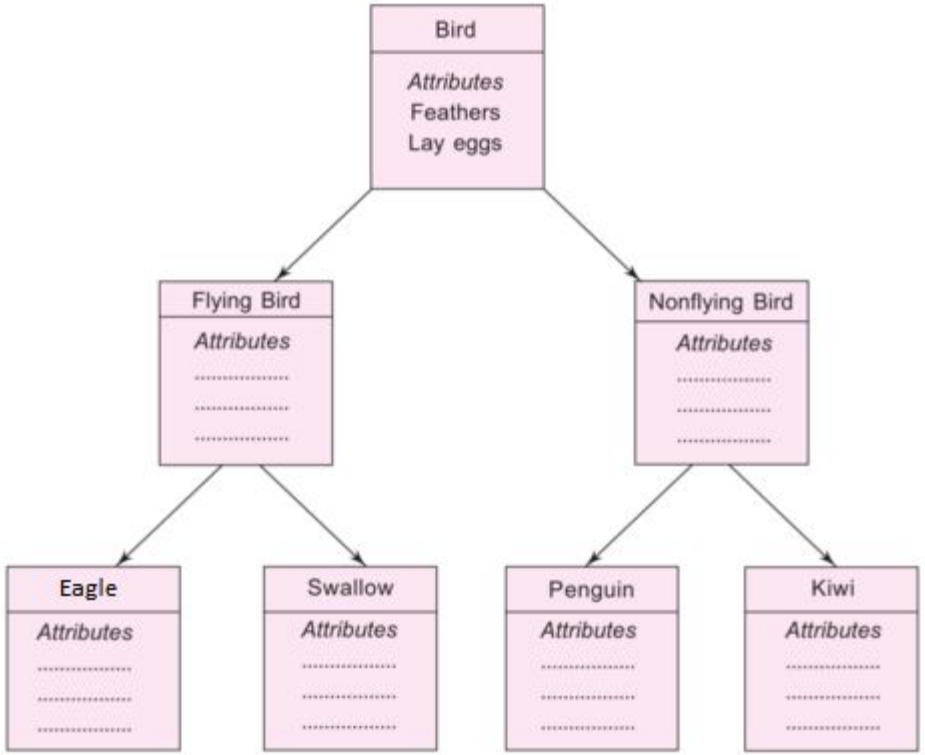
- *Point* with attributes *x* and *y*
- *Line* which contains two object of *Point* class
- *ColorLine* with attributes *Color* which extends *Line* class.

Override method *toString*() and define method *print*() in every classes

In *main*() method create array of *Line* and add some *Line* and *ColorLine* to it. Call method *print*() for all of it.

Homework

1. Develop abstract class *Bird* with attributes *feathers* and *layEggs* and an abstarct method *fly*().
   Develop classes *FlyingBird* and *NonFlyingBird*.
   Create class Eagle, Swallow, Penguin and Chicken.
   Create array *Bird* and add different birds to it.
   Call *fly()* method for all of it. Output the information about each type of created bird.



softserve

Homework

2. Support we have a class Employee

Create a Developer class that extends the Employee class. Creates a String field and a constructor to initialize all fields in the Developer class.

```
class Employee {
    private String name;
    private int age;
    private double salary;

    public Employee(String name, int age, double salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    public String report() {
        return String.format("Name: %s, Age: %d, Salary: \u20B4 %.2f.",
                name, age, salary);

    }
}
```

Also in the Developer class, override the method report() so that it returns a string with information about the developer, for example:

Name: Taras, Age: 32 years, Position: Average Java developer, Salary: 32735.35

If necessary, modify the employee's class so that it meets the principles of encapsulation and inheritance. Create an instance of the Employee and Developer class and print in the console information about them using report() method.

softserve

THANKS

softserve