

Object Oriented Programming

514770-1
Fall 2021
9/6/2021
Kyoung Shin Park
Computer Engineering
Dankook University

Software Crisis

- SW Crisis
 - The term “**software crisis**” was coined by Fridrich L. Bauer at the first NATO Software Engineering Conference in 1968.
 - The **causes** of the software crisis is related to
 - the **overall complexity of the software development process**
 - and the relatively **immaturity of software engineering**.
 - The crisis manifested itself in several ways
 - Projects running over-budget
 - Projects running over-time
 - Software was very inefficient
 - Software was of low quality
 - Software often did not meet requirements
 - Projects were unmanageable and code difficult to maintain
 - Software was never delivered

Software Quality

- Comparison with Architecture
 - Assuming you are building a house, how do you measure the quality?
 - What if one room came out less than you planned?
 - This is a significant design flaw.
 - Even if you made everything according to the design,
 - What if you can't meet the moving-in date?
 - What if the cost is more than your planned budget?
 - Design compliance is important, but delivery and cost are also important

Software Quality

- Software Quality
 - Quality problems fail to solve the set of requirements with the given amount of time with given amount of efforts
 - Time
 - Should not exceed the time limit
 - Effort
 - Measured in man-month
 - Same as time and cost
 - Requirements
 - **Functions that the users want**
 - Not what the developers want

Software Severity

- As software becomes larger, the crisis becomes more serious
 - Assuming a large enterprise program as 10,000,000 LOC
 - It's about 200,000 pages where 1 page contains 50 lines
 - It's about 700 books where 1 book contains 300 pages
 - The development cost is 300 billion won if 1 LOC costs 30,000 won
 - This cost including requirement analysis, design, testing, documentation, and inspection
- Problems in large-scale software development
 - Problems of collaboration
 - The importance of design
 - Divide, develop, and integrate large-scale software development
 - Problems of requirement
 - Requirements continue to grow

Software Severity

- The severity of the software error
 - There is a tendency not to value the severity of the software error
 - Software errors may threaten human life in medical and military fields
 - Software errors may cause financial losses in bank and financial sectors
 - Accident of 2016 Tesla autonomous driving

차량의 자동주행센서가 밝게 빛나는 하늘과 트럭의 흰색 면을 미처 구분하지 못한 것으로 테슬라 측은 파악하고 있다.



<https://www.mk.co.kr/news/special-edition/view/2016/07/473465/>

자율주행 첫 사망사고 충격...센서만으론 한계 드러낸 무인차

밝은 하늘과 흰색 트레일러 분간 못해 '팔' "무인차 상용화 연기" 신중론자 목소리 커져
자랑사고때 법적책임 문제도 다시 불거질듯

방인혁, 이진영 기자 | 입력 : 2016.07.01 15:52:01 수정 : 2016.07.06 16:20:15

◆ 자율주행차 첫 사망 사고 / 테슬라 모델S...美당국 조사 중 ◆

미국에서 자율주행 모드로 달리던 테슬라 모델S가 대형 트럭과 충돌해 운전자가 사망하는 사고가 발생했다. 전 세계에서 처음으로 발생한 자율주행차 사망 사고다. 테슬라는 미국 도로교통안전국(NHTSA)에 지난 5월 7일 플로리다에서 발생한 자율주행차 사망 사고 내용을 통보했고 NHTSA는 곧바로 사고 원인에 대한 조사에 들어갔다.

Good Software?

- User-centered design is a key concept in software development
 - Functionality, Efficiency, Maintainability, Reusability, Readability, etc are important
 - It is important to implement all the functions that users need, but it should be **easy to maintain, easy to reuse, and easy to read**
 - Maintenance costs will be increased if the software is not made easy to read
- Good Software
 - Easy to edit code because there is no code duplication and easy to understand
 - Convenient for others to take

Object-Oriented Programming

- Paradigm changes as a way to solve the SW crisis.
 - Changes towards faster and better to reduce software development time and cost → **Paradigm shift to OOP**
- Must understand the characteristics of object oriented programming
 - What is different compared to non-OOP?
 - Understanding what improves programming
 - Understanding inheritance as a way to reuse common parts
 - Distinguish when to use and when not to use

Object-Oriented Programming

- What is Object-oriented programming?
 - OOP is one of the programming styles
 - OOP improves the problems of procedural programming (or structured programming)
 - OOP consists of two elements (data and code)
 - Data
 - Value used for I/O and used while the code is executed
 - Code
 - Commands run by the computer
 - Use data to solve problems and produce results

Procedural Programming

- Problems of procedural programming
 - Separate procedures and data
 - Example: Gasoline car
 - The **move()** expresses **the process of moving a car**. The fuel enters the car engine and burns it to move the car. The energy generated in this process is transferred to the wheels to move the car.
 - You only need to call the **move()** when moving the car.
 - The **move()** needs **data** called **fuel**, but it cannot be put in the function.
 - The solution is **passed as a parameter**.

Problem with Procedural Programming

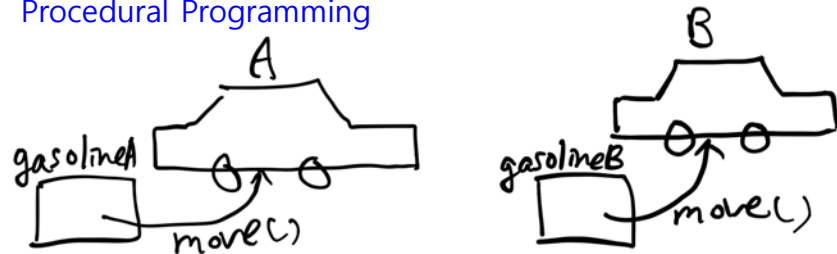
```
void move(Car car, double gas) {  
    // Use the given gas  
    // burn the gas in the engine  
    // transmit power to the wheel  
    // to move the car  
    ...  
}  
  
double gasoline = 20.0;  
move(A, gasoline);
```

Problem with Procedural Programming

- What if there are two cars?

```
double gasolineA = 20.0;
double gasolineB = 20.0;
move(A, gasolineA);
move(B, gasolineB);
```

Procedural Programming



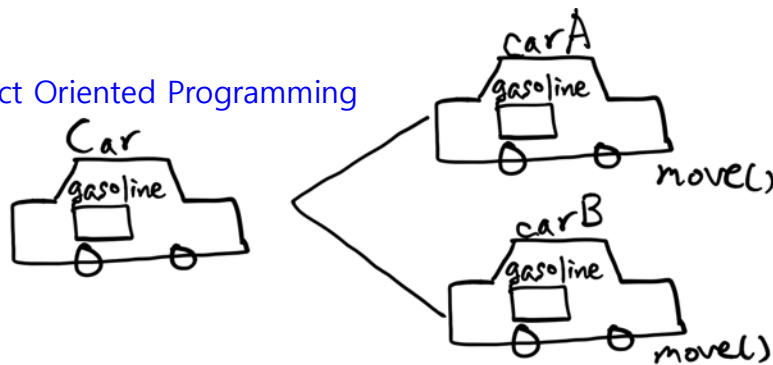
Object-Oriented Programming

- In object-oriented programming, data and procedures are grouped together using classes, and treated as a single data type.

```
class Car {
    double gasoline;
    void move() {
        ...
    }
}
Car carA = new Car();
Car carB = new Car();
carA.move();
carB.move();
```

Object-Oriented Programming

Object Oriented Programming



- The advantage of OOP is the reusability and readability of code.

OOP Concepts

- Object – objects have states and behaviors
- Class – defines the grouping of data and code, the “type” of an object
- Instance – specific allocation of a class
- Abstraction – hide the internal implementation of the feature, and only show the functionality to the user
- Encapsulation – keep implementation private and separate from interface
- Inheritance – hierarchical organization, code reusability, customize or extend behaviors
- Polymorphism – process objects differently based on their data type, using same interface

Abstraction

- Abstraction
 - Hide the underlying complexity of data
 - Help avoid repetitive code
 - Present only the signature of internal functionality
 - Give flexibility to programmers to change the implementation of the abstract behavior
 - Partial abstraction (0~100%) can be achieved with **abstract classes**
 - Total abstraction (100%) can be achieved with **interfaces**

Encapsulation

- Encapsulation
 - Restrict direct access to data members (fields) of a class
 - **Fields** are set to **private**
 - Each field has a **getter** and **setter** method
 - Getter methods return the field
 - Setter methods let us change the value of the field

Inheritance

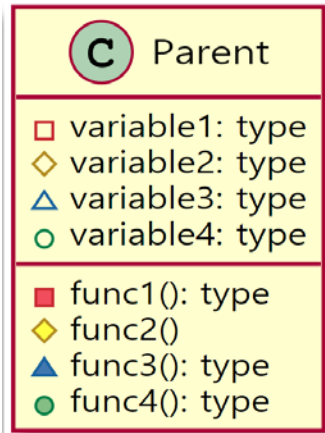
- Inheritance
 - A class (child class) can **extend** another class (parent class) by inheriting its features
 - Implement the **DRY (Don't Repeat Yourself)** programming principle
 - Improves **code reusability**
 - Multilevel inheritance is allowed in Java (a child class can have its own child class as well)
 - **Multiple inheritances are not allowed in Java** (a class can't extend more than one class)

Polymorphism

- Polymorphism
 - Polymorphism means existing in many forms
 - It means objects of different types can be accessed through the same interface. Each type can provide its own, independent implementation of this interface.
 - All Java objects can be considered polymorphic (at the minimum, they are of their own type and instances of the Object class)
 - Polymorphism could be static and dynamic.
 - Example of static polymorphism in Java is method overloading.
 - Example of dynamic polymorphism in Java is method overriding

UML Class Diagram

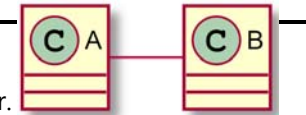
- Divide into three areas(class name, member fields, member methods)



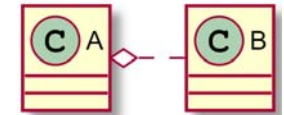
Character	Icon for field	Icon for method	Visibility
-	□	■	private
#	◇	◆	protected
~	△	▲	package private
+	○	●	public

UML Class Diagram

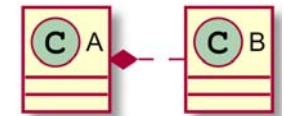
- Association
 - A and B class are associated with each other.
 - **Aggregation** and **Composition** are subsets of **Association**, i.e., specific cases of Association.



- Aggregation
 - Aggregation implies a relationship where the child can exist independently of the parent.



- Composition
 - Composition implies a relationship where the child cannot exist independent of the parent.
 - When A is deleted, then B is also deleted as a result.



UML Class Diagram

- Association

```
public class A { // A uses B
    void test(B b) { }
}
```

- Aggregation

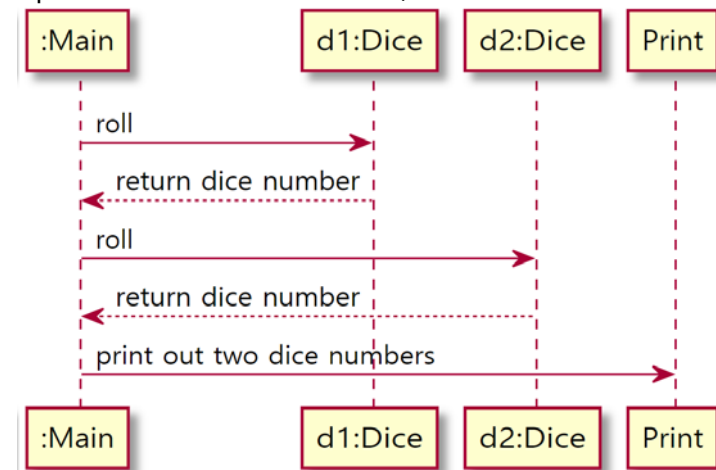
```
public class A { // When A dies, B may live on
    private B b;
    A(B b) { this.b = b; }
}
```

- Composition

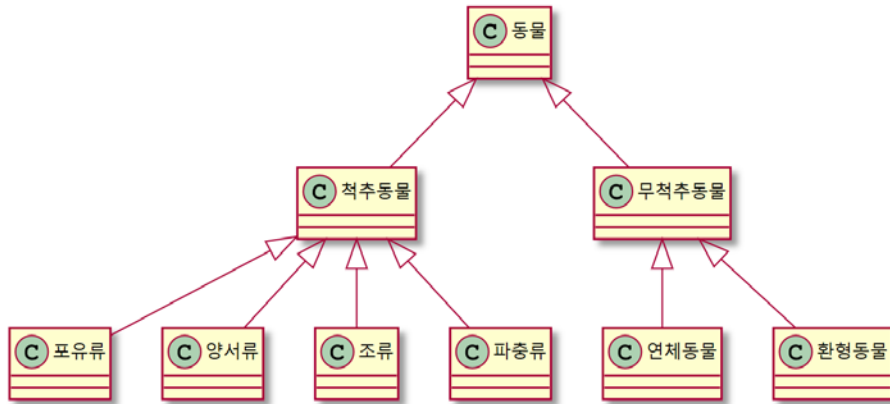
```
public class A { // When A dies, so does B
    private B b = new B();
}
```

UML Sequence Diagram

- Sequence Diagram – interaction diagram that details how operations are carried out (the order of the interaction)

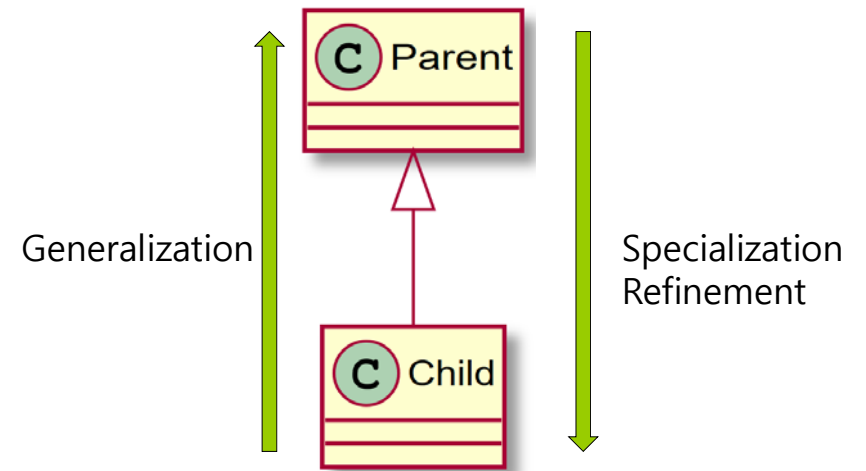


Inheritance



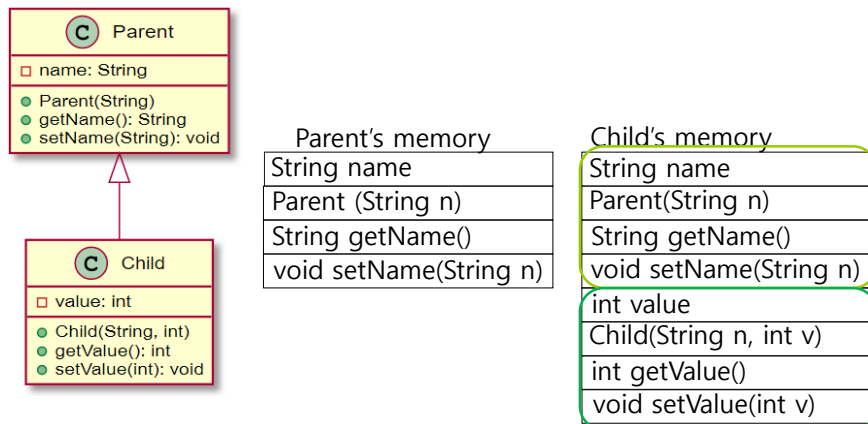
Inheritance

- Inheritance - extend, specialization



Inheritance

- A constructor cannot be inherited in Java.

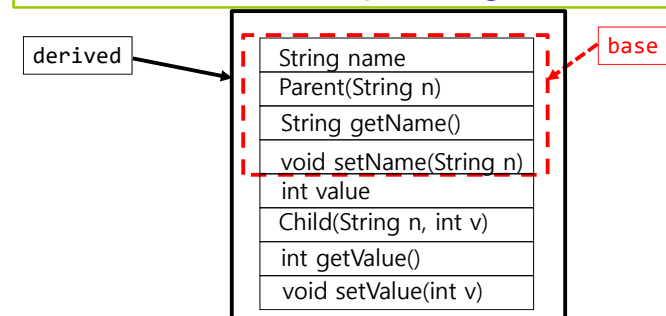


Inheritance

- You can store child object in parent reference (**upcasting**)
- Then, **you can only access the parent members** but it's not possible to access the child members.

```

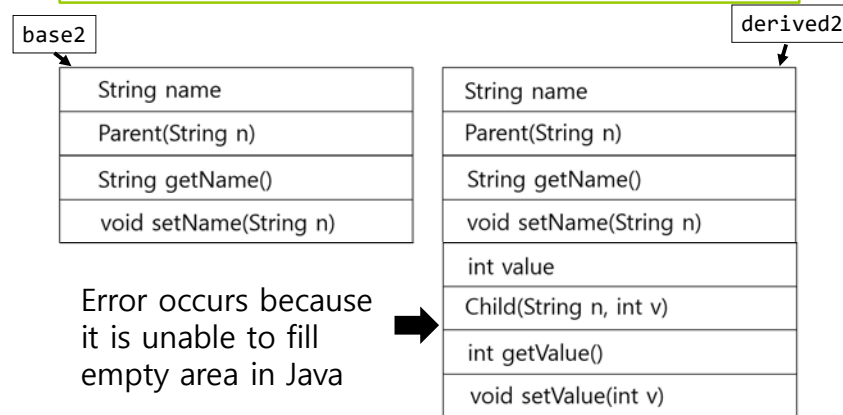
Parent base;
Child derived = new Child("park", 2020);
base = derived; // upcasting
    
```



Inheritance

- Downcasting **cannot** be done **implicitly**.

```
Parent base2 = derived;  
Child derived2 = base2; // compile error
```



Inheritance

- Downcasting means **typecasting** of a parent object to a child object. However, the original object must be a child.

```
Child derived2 = (Child)base2; // downcasting  
System.out.println(derived2.getValue());  
Parent base3 = new Parent("park");  
Child derived3 = (Child)base3; // error
```

- instanceof**

- If the left reference is the right class (or subclass) object, then it returns true, otherwise it returns false.
- If there is an inheritance relationship, the child class object is also identified as an object of the parent class.

```
if (derived2 instanceof Child) // true  
if (derived2 instanceof Parent) // true
```

Method Overloading

- Method Overloading

- Different number or types of function parameters
- Return type is meaningless
- Valid in the same class or classes with inheritance relationship

```
void print() { ... } // method overloading  
void print(String s) { ... } // method overloading  
void print(int n) { ... } // method overloading  
void print(String s, int n) { ... } // method overloading  
int print(int n) { ... } // error
```

Method Overriding

- Method Overriding

- The function's signature is the same
- Only meaningful in inheritance relationships

```
class Parent {  
    void print() { ... }  
    void print(String s, int n) { ... }  
}  
class Child extends Parent {  
    void print() { xxx } // method overriding  
    void print(String s, int n) { xxx } // method overriding  
}
```


Interface vs Abstract Class

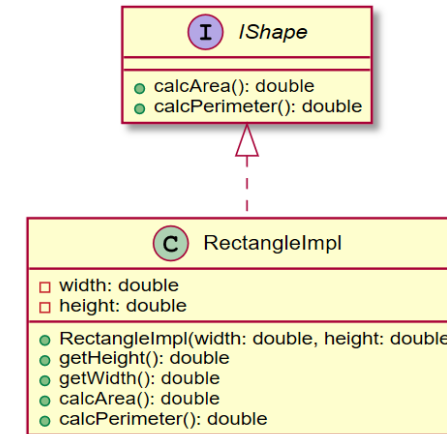
□ Interface

- Can have **constants and abstract methods** (only the function signature of the class to be implemented)
- From Java 8, it can have default and static methods (pre-implemented function)
- Class implementing interface must implement those that have only function signature.
- Members of a Java interface are public by default

□ Abstract Class

- Has **one or more abstract methods** (only the function signature)
- Can have member fields
- Can have abstract and non-abstract methods
- Can have class members like private, protected, public, default

Interface



Interface

```
// IShape.java
interface IShape {
    double calcArea();
    double calcPerimeter();
}

// RectangleImpl.java
class RectangleImpl implements IShape {
    private double width, height;
    public RectangleImpl(double width,
                          double height) {
        this.width = width;
        this.height = height;
    }
}
```

Interface

```
@Override
public double calcArea() {
    return width * height;
}

@Override
public double calcPerimeter() {
    return 2 * (width + height);
}

public double getHeight() { return height; }
public double getWidth() { return width; }
}
```

Interface

```
// RectangleMain.java
class RectangleMain {
    public static void main(String[] args) {
        IShape r = new RectangleImpl(10., 20.);
        System.out.println(r.calcArea());
    }
}
```

Interface – default method

```
//IValue.java
interface IValue {
    default int getValue() { return 0; }
}
// ValueImpl1.java
class ValueImpl1 implements IValue {
    private String name = "ValueImpl1";
    ValueImpl1(String s) { name = s; }
    public String getName() { return name; }
    public void setName(String s) {
        name = s;
    }
}
```

Interface – default method

```
// ValueImpl2.java
class ValueImpl2 implements IValue {
    private String name;
    ValueImpl2() {
        name = "ValueImpl2";
    }
    public String getName() { return name; }
    public void setName(String s) {
        name = s;
    }
    public int getValue() { return 1; } // default
    // method overriding
}
```

Interface – default method

```
// ValueMain.java
class ValueMain {
    public static void main(String[] args) {
        ValueImpl1 v1 = new ValueImpl1("ValueImpl1");
        ValueImpl2 v2 = new ValueImpl2();
        System.out.println(v1.getName());
        System.out.println(v2.getName());
        IValue i1 = v1;
        IValue i2 = v2;
        System.out.println(i1.getValue()); // 0
        System.out.println(i2.getValue()); // 1
    }
}
```

Abstract Class

```
// Shape.java
abstract class Shape {
    public abstract double calcArea();
    public abstract double calcPerimeter();
}
// RectangleImpl.java
class Rectangle extends Shape {
    private double width, height;
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
}
```

Abstract Class

```
@Override
public double calcArea() {
    return width * height;
}
@Override
public double calcPerimeter() {
    return 2 * (width + height);
}
public double getHeight() { return height; }
public double getWidth() { return width; }
}
```

Abstract Class

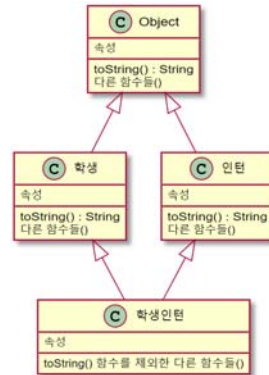
```
public class Circle extends Shape {
    private double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    @Override
    public double calcArea() {
        return Math.PI * radius * radius;
    }
    @Override
    public double calcPerimeter() {
        return 2 * Math.PI * radius;
    }
    public double getRadius() { return radius; }
}
```

Abstract Class

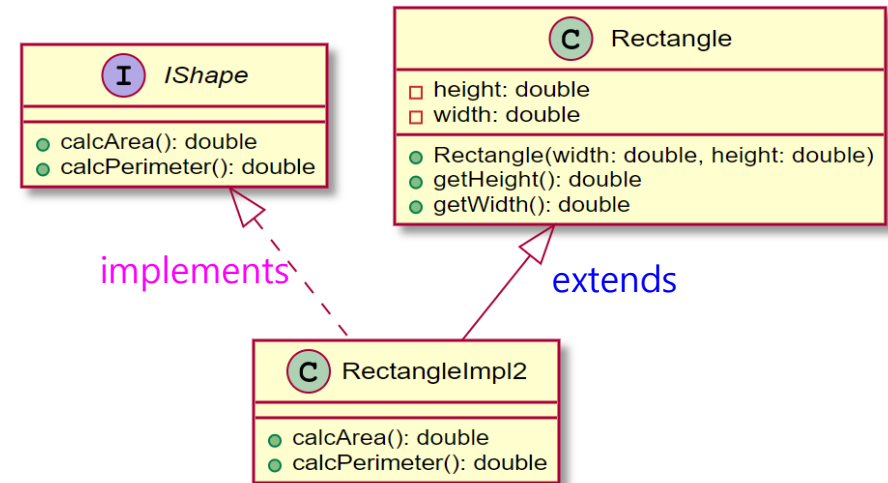
```
// AbstractShapeMain.java
class AbstractShapeMain {
    public static void main(String[] args) {
        Shape r = new Rectangle(20.0, 10.0);
        Shape c = new Circle(10);
        System.out.printf("Rectangle area:
%.2f\n", r.calcArea()); // dynamic binding
        System.out.printf("Circle perimeter:
%.2f\n", c.calcPerimeter()); // dynamic binding
    }
}
```

Multiple Inheritance

- The “**diamond problem**” is an ambiguity that can arise as a consequence of allowing multiple inheritance.
- Java does not allow multiple inheritance.
- Use interfaces instead of classes to achieve the same purpose



Multiple Inheritance



Multiple Inheritance

```
interface IShape {
    double calcArea();
    double calcPerimeter();
}
class Rectangle {
    private double width, height;
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
    public double getHeight() { return height; }
    public double getWidth() { return width; }
}
```

```
// RectangleImpl2.java
class RectangleImpl2 extends Rectangle
    implements IShape {
    public RectangleImpl2(double width,
        double height) {
        super(width, height);
    }
    @Override
    public double calcArea() {
        return getWidth() * getHeight();
    }
    @Override
    public double calcPerimeter() {
        return 2 * (getWidth() + getHeight());
    }
}
```

Multiple Inheritance

```
// RectangleMain.java
class RectangleMain {
    public static void main(String[] args) {
        RectangleImpl2 r = new RectangleImpl2(10, 10);
        Rectangle r2 = r;
        System.out.println(r2.getHeight());
        IShape s = r;
        System.out.println(s.calcArea());
    }
}
```

Multiple Inheritance – Interface default method

```
interface IValue {
    default int getValue() { return 0; }
}
class ValueImpl {
    private String name = "ValueImpl";
    ValueImpl() { }
    ValueImpl(String s) { name = s; }
    public String getName() { return name; }
    public void setName(String s) {
        name = s;
    }
}
```

Multiple Inheritance – Interface default method

```
// ValueImpl1.java
class ValueImpl1 extends ValueImpl implements IValue {
    ValueImpl1(String s) {
        super(s);
    }
}
// ValueImpl2.java
class ValueImpl2 extends ValueImpl implements IValue {
    ValueImpl2() {
        super();
        setName("ValueImpl2");
    }
    public int getValue() { return 1; } // default
    // method overriding
}
```

Multiple Inheritance – Interface default method

```
// ValueMain.java
class ValueMain {
    public static void main(String[] args) {
        ValueImpl1 v1 = new ValueImpl1("ValueImpl1");
        ValueImpl2 v2 = new ValueImpl2();
        System.out.println(v1.getName());
        System.out.println(v2.getName());
        IValue i1 = v1;
        IValue i2 = v2;
        System.out.println(i1.getValue()); // 0
        System.out.println(i2.getValue()); // 1
    }
}
```

Polymorphism

□ Polymorphism

- Allow us to perform a single action in different ways
- Execute specialized actions based on its type
- Call overridden methods in child classes from the parent reference variable at run time (**dynamic binding**)

```
public class ShapeTag {
    private String tag;
    public ShapeTag(String tag) {
        this.tag = tag;
    }
    public String toString() { return "#" + tag; }
}
```

```
public class RectangleTag extends ShapeTag {
    private String rectangleTag;
    public RectangleTag(String tag,
                        String rectangleTag) {
        super(tag);
        this.rectangleTag = rectangleTag;
    }
    @Override // Object toString method overriding
    public String toString() {
        return "#" + rectangleTag + " "
            + super.toString();
    }
    public String getRectangleTag() {
        return rectangleTag;
    }
}
```

```
public class CircleTag extends ShapeTag {
    private String circleTag;
    public CircleTag(String tag,
                    String circleTag) {
        super(tag);
        this.circleTag = circleTag;
    }
    @Override // Object toString method overriding
    public String toString() {
        return "#" + circleTag + " "
            + super.toString();
    }
    public String getCircleTag() {
        return circleTag;
    }
}
```

Polymorphism

```
ShapeTag s1 = new ShapeTag("shape1");
ShapeTag s2 = new ShapeTag("shape2");
RectangleTag r =
    new RectangleTag("shape", "rectangle");
CircleTag c = new CircleTag("shape", "circle");
System.out.println("Shape1 Tag: " + s1);
System.out.println("Shape1 Tag: " + s2);
System.out.println("Rectangle Tags: " + r);
System.out.println("Circle Tags: " + c);
```

```
Shape1 Tag: #shape1
Shape2 Tag: #shape2
Rectangle Tags: #rectangle #shape
Circle Tags: #circle #shape
```

Polymorphism

```
s1 = r; // upcasting
s2 = c; // upcasting

System.out.println("Rectangle Tags: " + s1); //
dynamic binding
System.out.println("Circle Tags: " + s2); //
dynamic binding
```

```
Rectangle Tags: #rectangle #shape
Circle Tags: #circle #shape
```

Polymorphism

```
ArrayList list = new ArrayList();
list.add(new ShapeTag("shape1"));
list.add(new ShapeTag("shape2"));
list.add(new RectangleTag("shape", "rectangle"));
list.add(new CircleTag("shape", "circle"));
for (Object o : list) {
    System.out.println(o); // dynamic binding
}
```

calcArea() using Polymorphism

- The area calculation method differs depending on the type of Shape.
- Comparison between non-OOP and OOP polymorphism
 - Version 1 – Use instanceof to classify the class object, and then call the calcArea()
 - Version 2 – Use polymorphism

```
Rectangle r = new Rectangle(3, 4);
Circle c = new Circle(5);
Shape[] shapes = new Shape[2];
shapes[0] = r;
shapes[1] = c;
```

calcArea() using Polymorphism

- Version 1: Use instanceof

```
for (Shape shape : shapes) {
    if (shape instanceof Rectangle) {
        Rectangle r = (Rectangle) shape;
        System.out.println(r.calcArea());
    }
    else if (shape instanceof Circle) {
        Circle c = (Circle) shape;
        System.out.println(c.calcArea());
    }
}
```

calcArea() using Polymorphism

- What if a new class called Triangle is added?

```
for (Shape shape : shapes) {
    if (shape instanceof Rectangle) {
        Rectangle r = (Rectangle) shape;
        System.out.println(shape.calcArea());
    }
    else if (shape instanceof Circle) {
        Circle c = (Circle) shape;
        System.out.println(c.calcArea());
    }
    else if (shape instanceof Triangle) {
        Triangle t = (Triangle) shape;
        System.out.println(t.calcArea());
    }
}
```

calcArea() using Polymorphism

- Version 2 – Use polymorphism

```
for (Shape shape : shapes)
    System.out.println(shape.calcArea());
```

- What if a new class called Triangle is added?

Generics

- ArrayList

- Data structure that allows access to elements using index, similar to array
- ArrayList is **dynamic in size**.
- ArrayList cannot contain primitive data types, and it can only contain objects
- You can insert elements into the ArrayList using add() method
- Length of the ArrayList is provided by the size() method

```
ArrayList list = new ArrayList();
list.add("Seoul");
list.add(new String("Tokyo"));
list.add(new Integer(3));
list.add(5); // auto-boxing
```

Generics

- Problem with non-generic ArrayList

- Can store any type of object
- Need typecasting (downcasting) when using the object
- Need to remember which element is which datatype is
- Mainly used for one data type

```
String s1 = list.get(0); // compile error -
                        // need typecasting
String s2 = (String) list.get(1);
String s3 = (String) list.get(2); // runtime
                        // exception
Integer i1 = (Integer) list.get(2);
int i2 = (Integer) list.get(3); // auto-
                        // unboxing
```


Generics

- Generics make errors to appear compile time than at run time.

```
ArrayList<String> list = new ArrayList<>();  
list.add("Seoul");  
list.add(new String("Tokyo"));  
list.add(new Integer(3)); // compile error  
String s = list.get(0); // no need for type  
casting
```

Generics

- Create a generic MyArrayList

```
class MyArrayList<E> {  
    ArrayList list;  
    public MyArrayList() {  
        list = new ArrayList();  
    }  
    public void add(E e) {  
        list.add(e);  
    }  
    public E get(int i) {  
        return (E) list.get(i);  
    }  
}
```

Generics

```
MyArrayList<String> l = new MyArrayList<>();  
l.add("temp");  
l.add("add");  
String s = l.get(0);
```