LESSON VIII. Polymorphism and generic programming

Vu Thi Huong Giang

SoICT (School of Information and Communication Technology)

HUST (Hanoi University of Science and Technology)

Objectives

- Master the polymorphism technique
- Understand the Java generic programming

Content

- Polymorphism
- Downcasting and upcasting
- Overloading
- Method call binding
- Generic programming

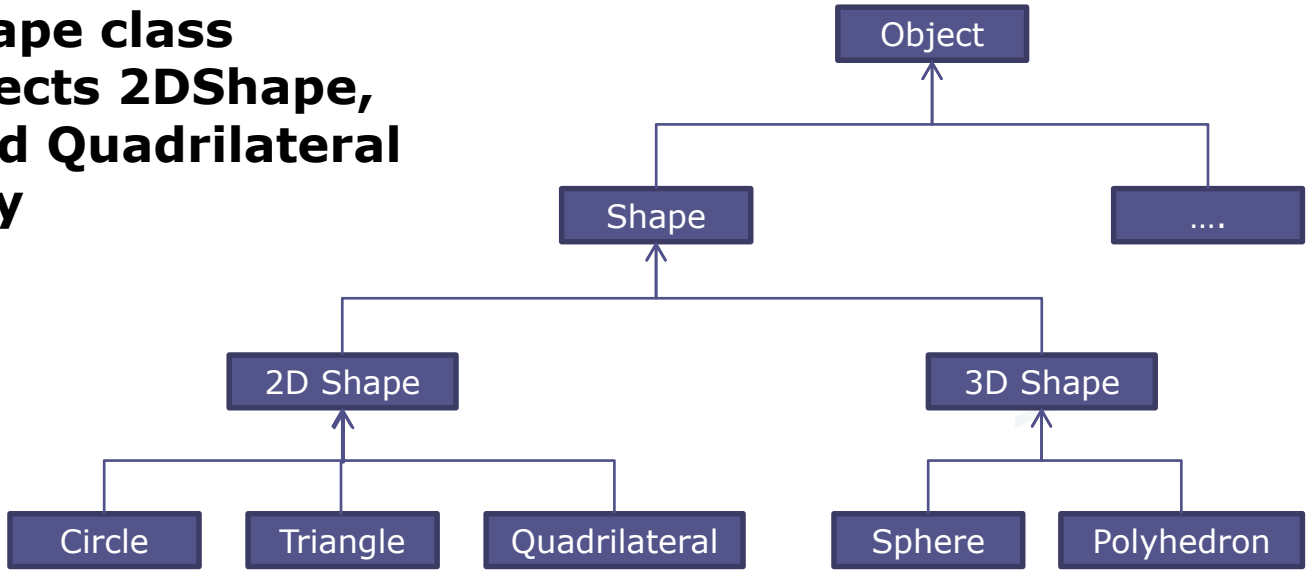
I. POLYMORPHISM

1. Example

Example

- An operation that can be performed on a 2DShape object can also be performed on an object of one of three classes Triangle, Circle, Quadrilateral.
  - The super class 2DShape defines the common interface
  - The subclasses Triangle, Circle, Quadrilateral have to follow this interface (inheritance), but are also permitted to provide their own implementations (overriding)

→ Once a method is requested through the 2DShape class reference, the objects 2DShape, Triangle, Circle and Quadrilateral respond differently



Example

```
public class 2DShape {
     public void display() {
        System.out.println("2D Shape");
     }
}
```

```
public class Point extends 2DShape {
     private int x, y;
     ...
     public void display(){
          System.out.print("(" + x + "," + y + ")");
     }
}
```

```
public class Circle extends 2DShape{
     public static final double PI =
        3.14159;
     private Point p;
     private double r; //radious

     ...
     public void display(){
          System.out.print("Circle: " +
             r + ",");
          p.display();
          System.out.println();
     }
}
```

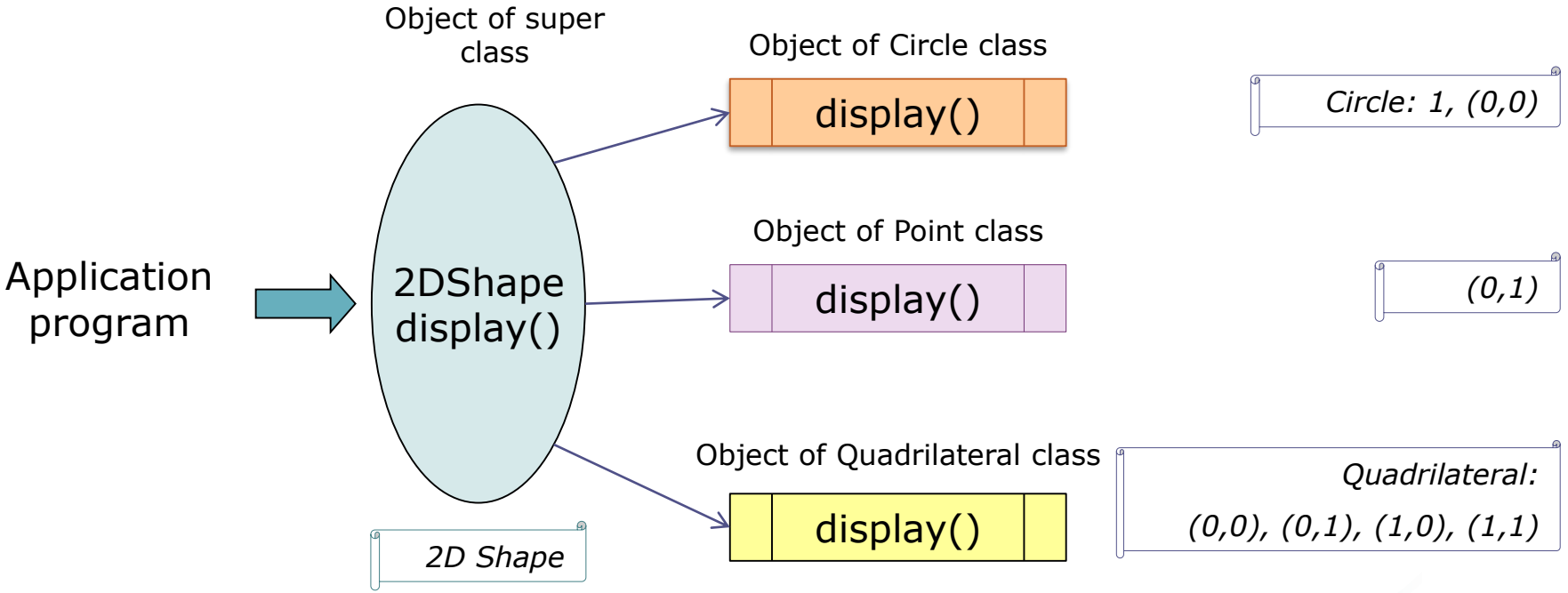
```
public class Quadrilateral extends 2DShape {
     private Point p1, p2, p3, p4;

     .....

     public void display(){
          System.out.println("Quadrilateral: ");
          p1.display(); p2.display();
          p3.display(); p4.display();
          System.out.println();
     }
}
```

Example

- There are many choice once a method is invoked through a super class reference.



Polymorphism

- Polymorphism means “many different forms” of objects
  - The ability of a reference variable to change behavior according to what object instance it is holding.
- Objects of different subclasses are treated as objects of a single super class
- Java choose the correct overridden method in the appropriate sub class associated with the object

Exercise

```
public class 2DShapeUsage {
     public static void main(String args[]) {
          Point point = new Point();
          Circle circle = new Circle();
          Quadrilateral quadri = new Quadrilateral();
          2DShape shape = new 2DShape();
          // display() method of 2DShape class is called
          shape.display(); // (1)
          // shape refers to the Point object; display() method of Point class is called
          shape = point; shape.display(); // (2)
          // display() method of Circle class is called
          shape = circle; shape.display(); // (3)
          // display() method of Quadrilateral class is called
          shape = quadri; shape.display(); // (4)

          2DShape shape2 = new Point();
          // display() method of Point class is called
          shape2.display(); // (5)
     }
}
```

II. DOWN CASTING AND UP CASTING

Primitive type casting

- Java performs automatic primitive type casting when:
  - Two types are compatible
  - The destination type is larger then the source type
  - Example:
    - int i;
    - double d = i;
- We have to perform manual primitive type casting when:
  - Two types are compatible
  - The destination type is smaller then the source type
  - Example:
    - int i;
    - ~~byte b = i;~~ byte b = (byte)i;

Reference type casting

- Java performs automatic reference type casting when:
  - Two types are **compatible**
  - The destination type extends from the source type

→ **Up casting**

- We have to perform manual reference type casting when:
  - two types are **compatible**
  - The source type extends from the destination type

→ **Down casting**

Up casting

- Substitute the reference of the sub class for the reference of the super class in the inheritance

```
public class 2DShape {
     public void display() {
        System.out.println("2D Shape");
     }
}
```

```
public class Point extends 2DShape {
     private int x, y;
     ...
     public void displayPoint(){
          System.out.print(“Point”);
     }
}
```

```
// I
...
Point point = new Point();
// 2 following statements are equivalent
2DShape shape = (2DShape) point;
2DShape shape = point;
```

```
// II
...
2DShape shape = new Point();
```

Up casting

- Java always remember what an object really is during up casting

```
public class 2DShape {
    public void display() {...}
}
```

```
public interface 2DShape {
    public void display();
}
```

```
public class Point extends 2DShape {
     public void displayPoint(){...}
}
```

```
public class Point implements 2DShape {
    // interface's methods
    public void display() {...}
    // class' methods
    public void displayPoint(){...}
}
```

```
...
Point point = new Point();
2DShape shape = point;
shape.display(); //OK
~~shape.displayPoint(); //impossible to call~~
```

```
...
Point point = new Point();
2DShape shape = point;
shape.display(); //OK
~~shape.displayPoint(); //impossible to call~~
```

Exercise: Implicit subclass object to super class object conversion

- Refer to a super class object with a super class reference
  - Example ?
- Refer to a sub class object with a subclass reference
  - Example ?
- Refer to a sub class object with a super class reference is safe, but such code can only refer to super class members
  - Example ?
- Refer to a super class object with a subclass reference is a syntax error
  - Example ?

Downcasting

- Substitute the reference of the super class for the reference of the sub class in the inheritance hierarchy.

→ May not always suceed

```
public class 2DShape {
     public void display() {
        System.out.println("2D Shape");
     }
}
```

```
public class Point extends 2DShape {
     private int x, y;
     ...
     public void displayPoint(){
          System.out.print(“Point”);
     }
}
```

```
...
2DShape shape = new Point();
Point point = (Point) shape; // 1
point.displayPoint(); //possible to call
```

```
...
Point point = new Point();
2DShape shape = point;
if (shape instanceof 2DShape){
   Point tempObj= (Point)shape;  // 2
   tempObj.displayPoint(); //possible to call
}
```

Type Compatibility

```
public class 2DShape {
      public void display() {
          System.out.println("2D Shape");
      }
}
```

- Does it work ?

```
...
Circle circle = new Circle();
2DShape shape = circle;
shape.changeWidth(20);
```

→ Syntax error

```
public class Circle extends 2DShape{
      public static final double PI = 3.14159;
      private Point p;
      private double r; //radious

      ...
      public void changeRadious(double rad){
            r = rad;
      }
}
```

- How can you make it work without changing the classes definitions?

```
...
Circle circle = new Circle();
2DShape shape = circle;
((Circle)shape).changeRadious(20); //ok
```

Problem with casting

```
public class 2DShape {
      public void display() {...}
}
```

```
public class Point extends 2DShape {
      private int x, y;
      ...
      public void displayPoint(){...}
}
```

```
public class Circle extends 2DShape{
      public static final double PI = 3.14159;
      private Point p;
      private double r; //radious

      ...
      public void changeRadious(double rad){...}
}
```

- Does it work ?

```
Point point = new Point();
Circle circle = new Circle();
TwoDimensionShape shape = circle;
((Point)shape).displayPoint();
```

→ Runtime exception

III. Method overloading

- Overloading defines two or more methods with the same name in a class.
- Overloaded methods are distinguished by their signature
  - same method name
  - different parameters or different number of parameters
- Example:

```
public class Point extends 2DShape {
     public void display(){
          System.out.print("(" + x + "," + y + ")");
      }

     public void display(int x) {
          System.out.print("x-coordinate: " + x);
     }
}
```

Example

```
public class Point extends 2DShape {
     // 1
     public void display(){
          System.out.print("(" + x + "," + y + ")");
      }

     // 2
     public void display(int x) {
          System.out.print("x-coordinate: " + x);
     }

     // 3- syntax error: Creating overloaded methods 2, 3 with identical parameters lists
     public void display(int y) {
          System.out.print("y-coordinate: " + y);
     }

     // 4- syntax error: Methods 1, 4 can not be distinguished by return type
     public boolean display() {
          System.out.print("x-coordinate: " + x);
          return true;
     }
}
```

Overloading constructors

- Overloading constructors allows creating objects in different ways

```
public class Circle extends 2DShape{
public static final double PI = 3.14159;
private Point p;
private double r; //radious
public Circle() {
     this.p = new Point(0,0);
     this.r = 1;
}
public Circle (Point p){
     this.p = p;
     this.r = 1;
}
public Circle (double r){
     this.p = new Point(0,0);
     this.r = r;
}
public Circle(Point p, double r){
     this.p = p;
     this.r = r;
}
```

```
// using overloaded constructors

public static void main( String[] args ){
     Point point = new Point();
     Circle circle1 = new Circle();
     Circle circle2 = new Circle(point, 2);
     Circle circle3 = new Circle(20);
     Circle circle4 = new Circle(point);

}
```

Overriding vs. Overloading

- Same signature
  - Identical method name
  - Identical parameter lists
  - Identical return type
- Defined in two or more classes related through the inheritance
- In a class: one overriding method per overridden method

- Different signatures
  - Identical method name
  - Different parameter lists (types, number)
  - Identical/ different return type (can not overload on return type)
- Defined in the same class
- In a class: several overloading methods per overloaded method

IV. METHOD CALL BINDING

1. Static binding
2. Dynamic binding

Static binding

- Method call is decided at compile-time
  - The called class and the calling class must be present at compile-time

→ This implies:

  - a static, non-extensible classing environment
  - functionality gets pushed higher and higher in the class hierarchy to make them available to more sub-classes

- Static or final method calls are resolved at compile-time.

Dynamic binding

- Method call is decided at run-time
- Method overriding allows dynamic binding:
  - An overridden method is called through the super-class variable
  - Java determines which version of that method to execute based on the type of the referred object at the time the call occurs
  - When different types of objects are referred, different versions of the overridden method will be called.
- Interfaces support dynamic binding.

V. GENERIC PROGRAMMING

What is generic programming ?

- Generic programming: creation of classes and methods that work in the same way on different types of objects
  - Generics with inheritance
  - Generics with type parameters: programming with classes and methods parameterized with types

Generic class

- Syntax

```
modifier class generic_class_name <type_param_1, .. type_param_n> {
        // instance variable
        // constructor
        // methods
}
```

- Example:

```
public class Information<T> {
        private T value;
                public Information(T value) {
                this.value = value;
        }
        public T getValue() {
                return value;
        }
}
```

Example: Type arguments

```
public class Information<T> {
     private T value;
          public Information(T value) {
          this.value = value;
     }
     public T getValue() {
          return value;
     }
}
```

```
....
// Can be instantiated with class or interface type:
Information<String> string = new Information<String>("hello"); //ok
Information<Circle> circle = new Information<Circle>(new Circle());
Information<2DShape> shape = new Information<2DShape>(new 2DShape());

// Cannot use a primitive type as a type variable
Information<int> integer = new Information<int>(2012); // failed
// Use corresponding wrapper class instead
Information<Integer> integer = new Information<Integer>(2012); //ok
```

Type parameter naming convention

Type Variable	Name Meaning
`E`	Element type in a collection
`K`	Key type in a map
`V`	Value type in a map
`T`	General type
`S, U`	Additional general types

Generic methods

- Method introducing its own type parameters
- Can be defined inside either generic or non-generic classes
- Can be either static or non static
- Syntax:

```
modifier <type_param1, …> return_type method_name(parameters_list) {
     …
}
```

- Example:

```
public <E> static void print(E[] a) {   … }
```

Example

```
public class ArrayTool {
// method, printing all elements of a string array
     public static void print(String[] a) {
          for (String e : a) System.out.print(e + " ");
          System.out.println();
     }
// generic method, printing all array elements of different types
     public static <E> void print(E[] a) {
        for (E e : a) System.out.print(e + " ");
        System.out.println();
     }
}
```

```
...
Point[] p = new Point[3];
String[] str = new String[5];
int[] intnum = new int[2];
ArrayTool.print(p);
ArrayTool.print(str);
// can not call generic method with primitive types
ArrayTool.print(intnum);
```

Bounded Type Parameters

- Bound: limits the parameter types that may be applied to a generic type
  - Class
  - Interface
- Single bound:

  <type_param extends bound>
- Multiple bounds:

  <type_param extends bound_1 & bound_2 & .. >

Example

```
public class Information<T extends 2DShape> {
    private T value;
    public Information(T value) {
        this.value = value;
    }
    public T getValue() {
        return value;
    }
}
```

```
...
Information<Point> pointInfo = new Information<Point>(new Point()); //OK

Information<String> stringInfo = new Information<String>(); // error
```

Example

```
public class ShapeInfo<T> {
      private T t;
      public void set(T t) {
            this.t = t;
      }
      public T get() {
            return t;
      }
      public <U extends 2DShape> void inspect(U u){
      System.out.println("T: " + t.getClass().getName());
            System.out.println("U: " + u.getClass().getName());
      }
}
```

```
...
ShapeInfo<Point> pointInfo = new ShapeInfo<Point>();
ShapeInfo<String> stringInfo = new ShapeInfo<String>();

pointInfo.set(new Point()); // OK
stringInfo.set(new Point()); // error: this is not a string

pointInfo.inspect(new Circle()); // OK
stringInfo.inspect(new Point()); // OK
pointInfo.inspect("some text"); // error: this is not a 2DShape
stringInfo.inspect("some text"); // erroe: this is not a 2Dshape
```

Type erasure: generics class

- Java compiler erases all type parameters and replaces each with:
  - Object (if the type parameter is unbounded)
  - its first bound (if the type parameter is bounded), or

```
public class ShapeInfo<T> {
      private T t;
      public void set(T t) {
            this.t = t;
      }
      public T get() {
            return t;
      }
}
```

```
public class ShapeInfo {
      private Object t;
      public void set(Object t) {
            this.t = t;
      }
      public Object get() {
            return t;
      }
}
```

```
public class Information<T extends 2DShape> {
      private T value;
      public Information(T value) {
            this.value = value;
      }
      public T getValue() {
            return value;
      }
}
```

```
public class Information {
      private 2DShape t;
      public void set(2Dshape t) {
            this.t = t;
      }
      public 2DShape get() {
            return t;
      }
}
```

Type erasure: generics method

- Java compiler erases all type parameters and replaces each with:
  - Object (if the type parameter is unbounded)
  - its first bound (if the type parameter is bounded), or

```
public static <E> void print(E[] a) {
   for (E e : a) System.out.print(e + " ");
   System.out.println();
}
```

```
public static void print(Object[] a) {
   for (Object e : a) System.out.print(e + " ");
   System.out.println();
}
```

```
public <U extends 2DShape> void inspect(U u){
         System.out.println("T: " + t.getClass().getName());
         System.out.println("U: " + u.getClass().getName());
}
```

```
public void inspect(2DShape u){
         System.out.println("T: " + t.getClass().getName());
         System.out.println("U: " + u.getClass().getName());
}
```

Wildcard types

- Wildcard (?) : unknown type

Name	Syntax	Meaning
Wildcard with lower bound	`? extends B`	Any sub type of `B`
Wildcard with higher bound	`? super B`	Any super type of `B`
Unbounded wildcard	`?`	Any type

Review

- Polymorphism:
  - multiple objects of different subclasses to be treated as objects of a single super class
- Type casting:
  - Up casting:sub class is type-cast to a super class
  - Down casting: super class is type-cast to a sub class
- Overloading: methods in the same class are distinguished by their signature
  - Overloading constructors: creating objects in different ways
- Method call binding:
  - Static binding: Method call is decided at compile-time
  - Dynamic binding: Method call is decided at run-time

Review

- Generic programming
  - Generic class / interface: parameterized over types
  - Generic method: introduce its own type parameters
  - Bound: constraint on the type of a type parameter
  - Type erasure: no new classes are created for parameterized types
    - Unbounded type parameters: replaced by Object
    - Bounded type parameters: replaced by bounds
  - Wildcard: unknown type of parameter, field, or local variable, unknown return type

Quiz

- Given 3 classes as follow:

```
public class 2DShape {
      public void toString() {...}
}
```

```
public class Point extends 2DShape {
      private int x, y;
      ...
      public void toString(){...}
}
```

```
public class Circle extends 2DShape{
      public static final double
         PI = 3.14159;
      private Point p;
      private double r; //radious
      ...
      public void toString(){...}
}
```

- Does the following code work ? Why ?

```
Circle c = new Circle(5);
Rect r = new Rect(5, 3);
Shape s = null;
if( Math.random(50) % 2 == 0 ) s = c;
else s = r;
System.out.println( "Shape is + s.toString());
```

Quiz

- Given 3 classes as follow:

```
public class 2DShape {
      public void toString() {...}
}
```

```
public class Point extends 2DShape {
      private int x, y;
      ...
      public void toString(){...}
}
```

```
public class Circle extends 2DShape{
      public static final double
         PI = 3.14159;
      private Point p;
      private double r; //radious
      ...
      public void toString(){...}
}
```

- The method toString() is overridden. Which version gets called ?

```
Circle c = new Circle(5);
Rect r = new Rect(5, 3);
Shape s = null;
if( Math.random(50) % 2 == 0 ) s = c;
else s = r;
System.out.println( "Shape is + s.toString());
```

Quiz

- Consider the following code

```
public class Pair <T, U> {
     public T first;
     public U second;
     public Pair (T x, U y) {
          first = x;
          second = y;
     }
     public Pair () {
          first = null;
          second = null;
     }
}
```

- Which one is correct ?

1. Pair pair = **new Pair<Integer, Integer>();**
2. Pair pair = **new Pair<Byte, byte>();**
3. Pair pair = **new Pair<int, Circle>(0, new Circle());**
4. Pair pair = **new Pair<Point, Circle>(new Circle());**

Quiz

- Which is the raw class of the following code:

```
public class Pair <T, U> {
     public T first;
     public U second;
     public Pair (T x, U y) {
          first = x;
          second = y;
     }
     public Pair () {
          first = null;
          second = null;
     }
}
```

- Answer

```
public class Pair {
     public Object first;
     public Object second;
     public Pair (Object x, Object y) {
          first = x;
          second = y;
     }
     public Pair () {
          first = null;
          second = null;
     }
}
```