JAVA

Lecture VII – Inheritance

OOP

- Four properties of OOP:
  - Encapsulation: Objects have a 'layer' around them that holds the data and methods in one logical place. E.g., use private instance variables.。
  - Polymorphism:
    - Overload: multiple methods/constructors with the same name but different signature.
    - Override.
  - Inheritance.
  - Abstraction.

EXAMPLE: 2D SHAPE

```
public class Elephant{
	String color = “Grey”;
	String size = “Large”;
	String name;

	public Elephant(String n){
		name = n;
	}

	public String toString(){
		return name + “ ” + color
+ ” ” + size;
	}
}
```

```
public class RoyalElephant{
	String color = “White”;
	String size = “Large”;
	String name;

	public RoyalElephant(String n){
		name = n;
	}

	public String toString(){
		return name + “ ” + color
+ ” ” + size;
	}

}
```

EXAMPLE: 2D SHAPE

```
public class Elephant{
	String color = “Grey”;
	String size = “Large”;
	String name;

	public Elephant(String n){
		name = n;
	}

	public String toString(){
		return name + “ ” + color
+ ” ” + size;
	}
}
```

```
public class RoyalElephant{
	String color = “White”;
	String size = “Large”;
	String name;

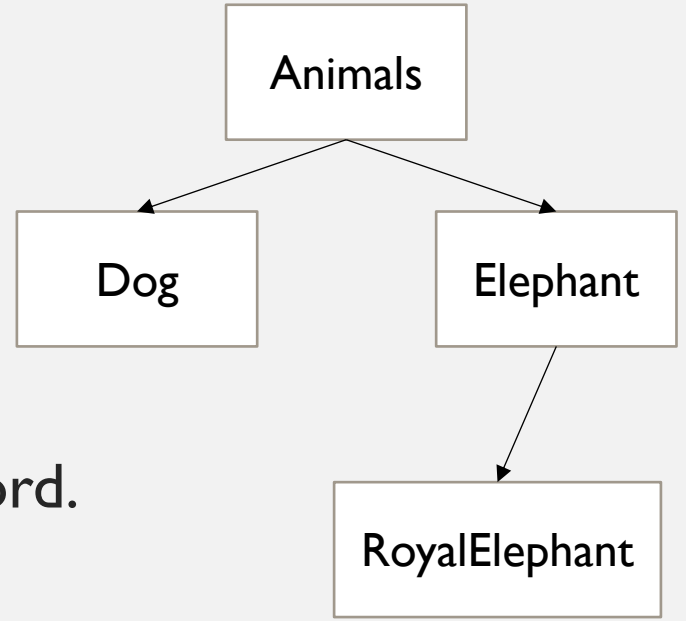
	public RoyalElephant(String n){
		name = n;
	}

	public String toString(){
		return name + “ ” + color
+ ” ” + size;
	}

}
```

INHERITANCE

- In Java, the data and methods are grouped as classes.
- We may have two similar classes, one is more general, the other is more specific. But both of them use similar data and methods.
- E.g., Elephant and RoyalElephant.
- The more general class is called "superclass".
- The more specific one is called "subclass".
- Subclass inherit all the data and methods from the superclass.
- The inheritance relationship is declared by the **extends** keyword.



EXAMPLE: 2D SHAPE

```
class TwoDShape{
      double width;
      double height;

      void showDim(){
            System.out.println("Width
            and height are " + width +
            " and " + height);
      }
}
```

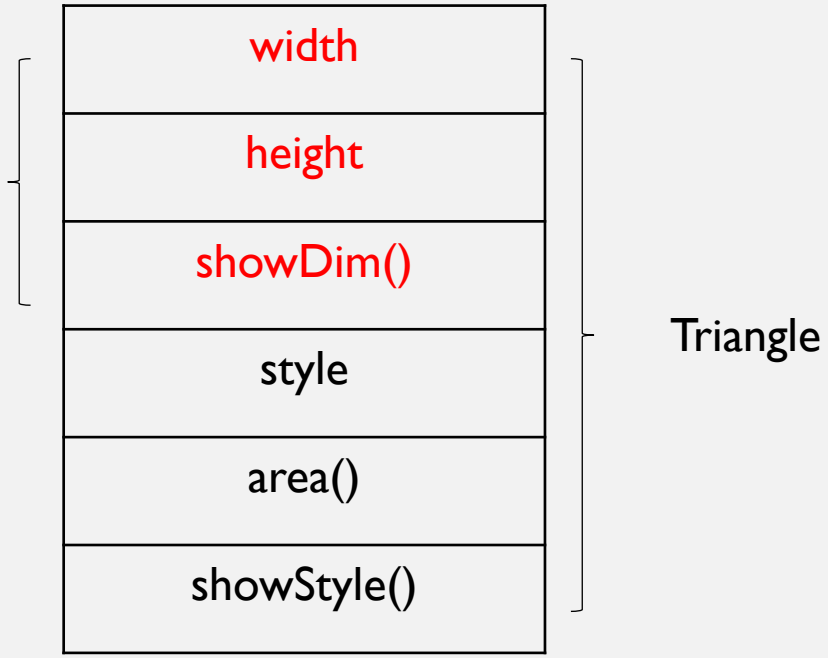
```
class Triangle extends TwoDShape{
      String style;

      double area(){
            return width * height / 2;
      }

      void showStyle(){
      System.out.println("Triangle is "
      + style);
      }
}
```

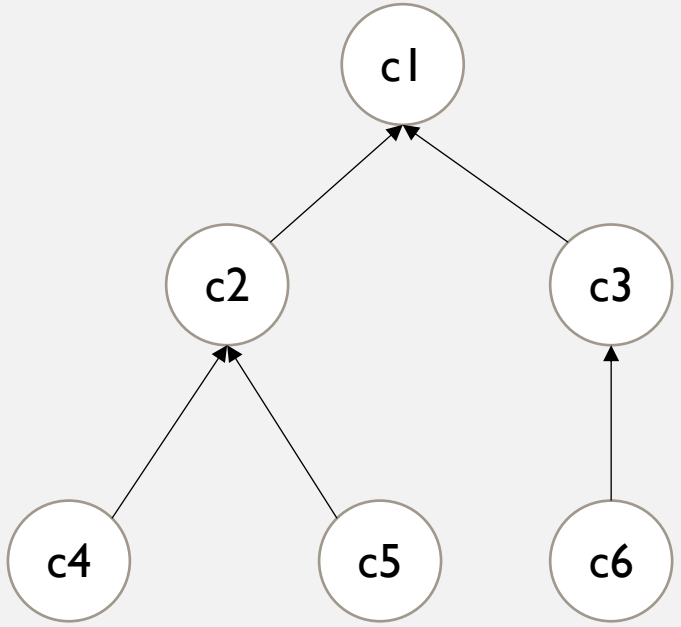
EXAMPLE: 2D SHAPE

- A subclass includes all of the members its superclass has.
- A subclass can have additional members.
- However, a super class has no knowledge of its subclasses.
- Each superclass can have multiple subclasses.
- But each subclass can have only one superclass (why?).
- Can subclasses access the private member of its superclass?



WHY INHERITANCE USEFUL

- Superclass: defines the common attributes.
- Subclass: contains specific methods and data.
- Inheritance creates a hierarchical structure.



- A special class “Object”
- toString()
- Equals(Object x)
- Hashcode()
- getClass()
- “” + a “” + a.toString();

WHAT ABOUT CONSTRUCTORS?

- Both a superclass and a subclass are objects, do they have constructors?

```
public class A{
      int value;
}
```

```
public class B extends A{
      B(){
            value = 1;
      }
}
```

- What if

```
A a = new A();
```

Which constructor will be used? What is the value of `a.value;`

WHAT ABOUT CONSTRUCTORS?

- Both a superclass and a subclass are objects, do they have constructors?

```
class A{
      int value;
      A(){
            value = 1;
      }

}
```

```
class B extends A{
}
```

- What if

```
B b = new B();
```

Which constructor will be used? What is the value of `b.value;`

WHAT ABOUT CONSTRUCTORS?

- Both a superclass and a subclass are objects, do they have constructors?

```
class A{
      int value;
      A(){
            value = 1;
      }

}
```

```
class B extends A{
      B(){
            value = 2;
      }

}
```

- What if

```
B b = new B();
```

Which constructor will be used? What is the value of `b.value;`

WHAT ABOUT CONSTRUCTORS?

- Both a superclass and a subclass are objects, do they have constructors?

```
class A{
      int value;
      A(){
            value = 1;
      }

}
```

```
class B extends A{
      B(){
            value = 1;
            value = 2;
      }

}
```

- What if

```
B b = new B();
```

Which constructor will be used? What is the value of `b.value;`

WHAT ABOUT CONSTRUCTORS?

- Both a superclass and a subclass are objects, so they have constructors?

```
class A{
	int value;
	A(int n){
		value = n;
	}

}
```

```
class B extends A{
	B(){
		value = 2;
	}

}
```

- What if

```
B b = new B();
```

Which constructor will be used? What is the value of `b.value;`

CALL SUPERCLASS CONSTRUCTORS

- How to call the constructor of its superclass?

```
super(parameter-list)
```

- super() must be called as the first statement.

```
class A{
      int value;
      A(int n){
            value = n;
      }
}
```

```
class B extends A{
      int value2;
      B(int n, int m){
            super(n);
            value2 = m
      }
}
```

SUPER KEYWORD

- ```
  class Parent{
      int i;
  }
  ```

```
class Child extends Parent{
    int i;
    Child(int n){
        i = n; // which i?
    }
}
```
```

SUPER KEYWORD

- What if how we want to change the value of i in Parent?

```
class Parent{
    int i;
}
```

```
class Child extends Parent{
    int i;
    Child(int n, int m){
        i = m;
    }
}
```

- Remember how **this** is used, can we do the following:

```
super.member
```

SUPER KEYWORD

- What if how we want to change the value of i in Parent?

```
class Parent{
    int i;
}
```

```
class Child extends Parent{
    int i;
    Child(int n, int m){
        super.i =n;
        i = m;
    }
}
```

SUPERCLASS AND SUBCLASS OBJECTS

- Java is a strongly typed language.

```
X x = new X();
X x2;
Y y = new Y();

x2 = x;  // will it work?
x2 = y;  // will it work?
```

SUPERCLASS AND SUBCLASS OBJECTS

- Java is a strongly typed language.

```
X x = new X();
X x2;
Y y = new Y();

x2 = x;  // will it work?
x2 = y;  // will it work?
```

- It works only when Y is a subclass of X.

SUPERCLASS AND SUBCLASS OBJECTS

- Java is a strongly typed language.

```
X[] xs   = new X[2];
xs[0] = x;
xs[1] = y;

Y y2 = xs[1];          // will it work?
xs[1].b = 27;          // will it work?
```

SUPERCLASS AND SUBCLASS OBJECTS

- Java is a strongly typed language.

```
X[] xs   = new X[2];
xs[0] = x;
xs[1] = y;

Y y2 = (Y) xs[1];          // will it work?
y2.b = 27;                 // will it work?

// to check the real type
if( xs[1] instanceof Y){
   Y y2 = (Y) xs[1];
}
```

OVERRIDING

- Polymorphism: overload and override.
- Method Overriding:
- When? a method is declared in both a superclass and a subclass. i.e., same name, return type and same input parameters.
- What happens? When this method is called from the subclass, it will refer to the one defined in the subclass.
- The decision is made at runtime.

EXAMPLE

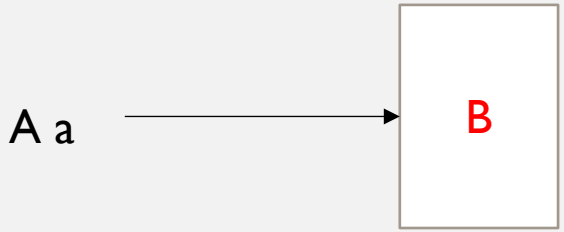
```
class Vehicle{
	public void move(){
		System.out.println(“All vehicles move”);
	}
}
class Boat extends Vehicle{
	public void move(){
		System.out.println(“Boats float”);
	}
}
```

EXAMPLE

```
class Vehicle{
	public void move(){
		System.out.println("All vehicles move");
	}
}
class Boat extends Vehicle{
	public void move(String s){
		System.out.println("Boats " + s);
	} // method override here?
}
```

OVERRIDING

- Dynamic method dispatch: a call to an overridden method is resolved at run time rather than compile time. (runtime polymorphism).
- A extends B
- `A a = new B();`
- `a.method()`
- When an overridden method is called through a superclass, Java determine which version of the object being referred to at run time. (i.e., the call determined by the type it refers to but not the type of the reference variable)



OVERLOADING VS OVERRIDING

When	Compile-time	Runtime
Where	Within a class	Two classes with inheritance relationship
Inheritance	Not necessary	Always required
Parameters	Must be different	Must be the same
Return type	Doesn't matter	Must be the same

STATIC VS NON-STATIC

- Static method: is called according to the type of class, not according to the type of object it refers to, i.e., decided at compile time.

- Non-static method: is called through the object, not according to the class type, i.e., decided at run-time

- Thus, we CANNOT override static method.

FINAL KEYWORDS

- How to prevent overriding? Use the keywords **final**.

```
class A{
        final void method(){} // this method cannot be overridden
}
```

- Moreover, **final** can be used to prevent inheritance, i.e., final class.
- All methods in a final class are declared (implicitly) as final
- **final** can also be applied to variables. (What does it means?)

ABSTRACTION

- Abstraction: provide high-level specifications but hide the detail implementations.
- Why we need abstractions?
- Abstraction: when a superclass is unable to create a meaningful implementation

- Example: 2D shape is an abstract concept, a method to calculate its area does not make any sense.
- However, we can have methods to calculate the area of a triangle.

- Both the classes and the methods can be abstract in Java.

ABSTRACT METHODS

- Abstraction: declare methods that must be overridden by its subclass.
- Keywords **abstract** is used to declare abstract classes and methods.
- Abstract method: only specify the return type, input parameters and the name.

```
abstract type name(parameter-list);
```

- Properties:
  - Abstract methods must be implemented in its subclass (True or false?)
  - Static methods can not be declared as abstract (Why?)
  - An abstract method can only be declared within an abstract class.

ABSTRACT CLASSES

- Abstract Class:
  - Can contain both abstract and concrete methods.
  - Cannot be instantiated (no object of such class can be generated).
  - When a class inherits an abstract class, it must implement all the abstract methods. (What happens if it does not)

```
abstract class TwoDShape{

      …

      abstract double area();

}
```

ABSTRACTION

- Abstract Classes: What a class must do
- Concreate Classes: How it will do it
- What are the problems of the abstract classes?

- Interface: fully separate the specification from the implementation.
  - Does not allow concreate methods.
  - An interface can be implemented by multiple classes.
  - One class can implement multiple interfaces, e.g., Comparable, Cloneable, …