

## Chapter 11 Inheritance and Polymorphism

Suppose you will define classes to model circles, rectangles, and triangles. These classes have many common features.

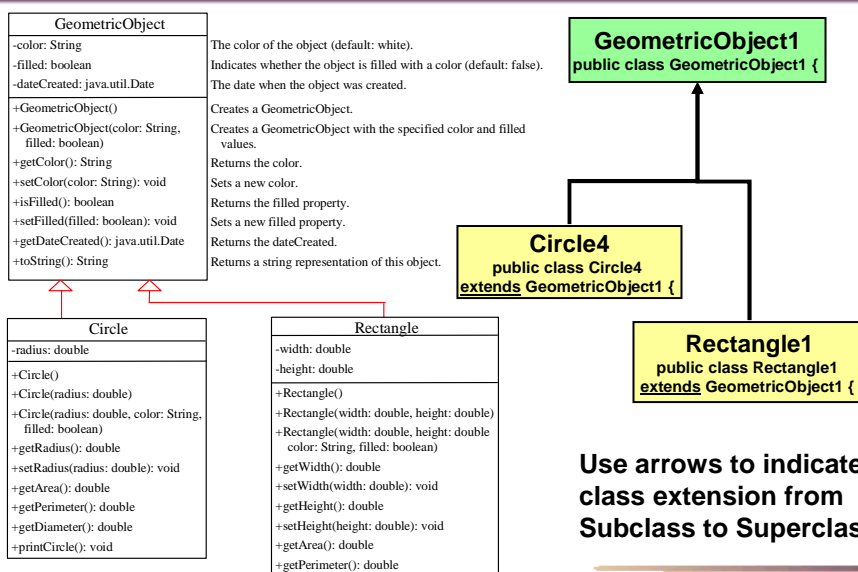
The best way to design these classes so to avoid *redundancy* (*duplicate code*) is to use *inheritance*.

### ❖ Inheritance

- ◆ Software reusability
- ◆ Create new class from existing class
  - ◆ Utilizes existing class's data and behaviors
  - ◆ Extends class with additional properties and methods
- ◆ Subclass extends superclass
  - ◆ Subclass
    - ▶ More specialized group of objects
    - ▶ Methods inherited from superclass
      - ↳ Can create additional methods in subclass
      - ↳ Can overwrite superclass methods in subclass
    - ▶ Properties inherited from superclass
      - ↳ Can create additional properties in subclass

1

## UML Superclasses and Subclasses



2

## Abstraction, Inheritance, Composition

### ❖ Abstraction

- ◆ Focus on commonalities among objects in system

### ❖ “is-a” vs. “has-a”

#### ◆ “is-a”

- ◆ Inheritance
- ◆ subclass object treated as superclass object
- ◆ Example: Car *is a* vehicle
  - Vehicle properties/behaviors also car properties/behaviors
- ◆ Dog is a animal
  - Animal properties/behaviors also dog properties/behaviors

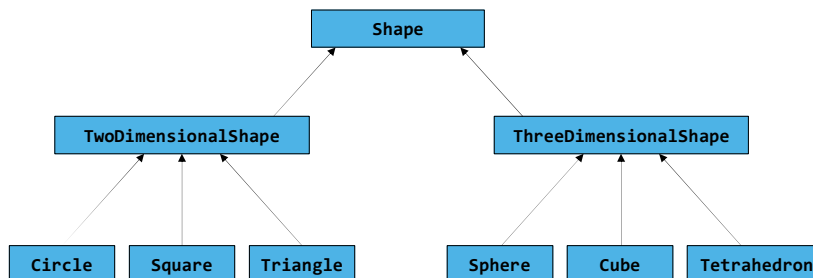
#### ◆ “has-a”

- ◆ Composition
- ◆ Object contains one or more objects of other classes as members
- ◆ Example: Car *has a* steering wheel
- ◆ Example: Kitchen *has a* refrigerator

3

## Subclass Inherits from Superclass

- ❖ Direct superclass
  - ◆ Inherited explicitly (one level up hierarchy)
- ❖ Indirect superclass
  - ◆ Inherited two or more levels up hierarchy
- ❖ Single inheritance
  - ◆ Inherits from one superclass
- ❖ Multiple inheritance
  - ◆ Java does not support multiple inheritance



4

## Examples of Subclass and Superclass

- ❖ Inheritance can be verified with "is-a" test
  1. Guitar extends (is-a) Instrument? Yes
  2. Person extends (is-a) employee? Rev.
  3. Dog extends (is-a) animal? Yes
  4. Metal extends (is-a) Aluminum? Rev.
  5. Oven extends Kitchen? Cmp.
  6. Ferrari extends engine? Cmp.
  7. Blond extends smart? No
  8. Coke extends Beverage? Yes

5

## The keyword super

- ❖ **super** refers to the superclass of the class (subclass) in which super appears
- ❖ **super** can be used in two ways:
  - ◆ To call a superclass constructor
  - ◆ To call a superclass method
- ❖ **Caution:** You must use the keyword **super** to call the superclass constructor
  - ◆ Invoking a superclass constructor's name in a subclass causes a syntax error
  - ◆ Java requires that the statement that uses the keyword **super** appear first in the constructor

6

## Instantiating Subclass Object

- ❖ Unlike properties and methods, a superclass's constructors are not inherited in the subclass they are automatically invoked
- ❖ Subclass constructor invokes Superclass constructor
  - ◆ Explicitly invoked from the subclasses' constructors, using the keyword `super`
  - ◆ Implicitly invoked if the keyword `super` is not explicitly used, the superclass's **no-arg constructor** is automatically invoked
    - ◆ If class does not have a no-arg constructor it results in error
  - ◆ Chain of constructor calls follows inheritance hierarchy
    - ◆ However, since constructor Last constructor called in chain is Object's constructor
    - ◆ Original subclass constructor's body finishes executing last
    - ◆ Example: Person :: Employee :: Faculty hierarchy
      - ▶ Constructor Invocation: Faculty → Employee → Person
      - ▶ Constructor executes last to first constructor in inheritance chain
      - ▶ Constructor code execution: Person → Employee → Faculty

7

## Constructor Chaining

Constructing an instance invokes all the superclasses' constructors along the inheritance chain

```

public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }
    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }
    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
    
```

The diagram illustrates the sequence of constructor calls during the instantiation of a Faculty object. It shows the following steps:

1. Start from the main method
2. Invoke Faculty constructor
3. Invoke Employee's no-arg constructor
4. Invoke Employee(String) constructor
5. Invoke Person() constructor
6. Execute println
7. Execute println
8. Execute println
9. Execute println

The class hierarchy is shown as follows:

- Person**: +Person()
- Employee**: +Employee(), +Employee(s: String)
- Faculty**: +Faculty()

Arrows indicate the inheritance chain: Faculty inherits from Employee, which inherits from Person.

8

## Superclass without no-arg Constructor

What is the program error:

```
public class Apple extends Fruit {  
}  
  
class Fruit {  
    public Fruit(String name) {  
        System.out.println("Fruit's constructor is invoked");  
    }  
}
```

9

## Overriding Methods in the Superclass

- ❖ A subclass inherits methods from a superclass
- ❖ A subclass extends properties and methods from the superclass

❖ You can also:

- ◆ Add new properties

- ◆ Add new methods

```
public void printCircle() {  
    System.out.println("The circle is created " +  
        super.getDateCreated() + " and the radius is " + radius);  
}
```

- ❖ Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass which is referred to as **method overriding**

```
public class Circle extends GeometricObject {  
    /** Override the toString method defined in GeometricObject */  
    public String toString() {  
        return super.toString() + "\nradius is " + radius;  
    }  
}
```

10

## Overriding Instance Methods and hiding static methods in the Superclass

An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden.

11

## Overriding vs. Overloading

### Overriding

```
public class Test {
    public static void main(String[] args) {
        A a = new A();
        a.p(10);
        a.p(10.0);
    }
}

class B {
    public void p(double i) {
        System.out.println(i * 2);
    }
}

class A extends B {
    // This method overrides the method in B
    public void p(double i) {
        System.out.println(i);
    }
}

    10.0
    10.0
```

### Overloading

```
public class Test {
    public static void main(String[] args) {
        A a = new A();
        a.p(10);
        a.p(10.0);
    }
}

class B {
    public void p(double i) {
        System.out.println(i * 2);
    }
}

class A extends B {
    // This method overloads the method in B
    public void p(int i) {
        System.out.println(i);
    }
}

    10
    20.0
```

12

## Polymorphism

- ❖ An object of a subtype (subclass) can be used wherever its supertype (superclass) value is required
  - ◆ Subtypes and Supertypes refer to class inheritance
  - ◆ Variable of a supertype can refer to a subtype object
  - ◆ Polymorphism allows code to run that works for class and any of its subclasses. Cool!

```

1. public class PolymorphismDemo {
2.     /** Main method */
3.     public static void main(String[] args)
4.     {
5.         // Display circle and rectangle properties
6.         displayObject(new Circle4(1, "red", false));
7.         displayObject(new Rectangle1(1, 1, "black", true));
8.     }
9.     /** Display geometric object properties */
10.    public static void displayObject(GeometricObject1 object)
11.    {
12.        System.out.println("Created on " + object.getDateCreated() +
13.            ". Color is " + object.getColor());
14.    }
15. }

```

Created on Tue Jun 12 00:07:28 JST 2012. Color is red  
Created on Tue Jun 12 00:07:28 JST 2012. Color is black

13

## Polymorphism and Dynamic Binding

```

public class DynamicBindingDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }

    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
}

class Student extends Person {
    public String toString() {
        return "Student";
    }
}

class Person extends Object {
    public String toString() {
        return "Person";
    }
}

```

Method `m` takes a parameter of the Object type. You can invoke it with any object.

An object of a subtype can be used wherever its supertype value is required. This feature is known as *polymorphism*.

When method `m(Object x)` is executed, the argument `x`'s `toString` method is invoked. `x` may be an instance of `GraduateStudent`, `Student`, `Person`, or `Object`.

Classes `GraduateStudent`, `Student`, `Person`, and `Object` have their own implementation of the `toString` method.

Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime.

This capability is known as *dynamic binding*.

Student  
Student  
Person  
java.lang.Object@7d8a992f

14

## Generic Programming

```
public class DynamicBindingDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }

    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
}

class Student extends Person {
    public String toString() {
        return "Student";
    }
}

class Person extends Object {
    public String toString() {
        return "Person";
    }
}
```

### ❖ Polymorphism


- ◆ Allows methods to be used generically for a wide range of object arguments.
- ◆ Is **Generic Programming**
- ◆ If a method's parameter type is a superclass (e.g., Object), you may pass an object to this method of any of the parameter's subclasses (e.g., Student or String).
- ◆ When an object (e.g., a Student object or a String object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., toString) is determined dynamically.

15

## Casting Objects

### ❖ Casting operator converts between primitive datatypes

### ❖ Casting can be used to convert between objects

- ◆ Classes must be within an inheritance hierarchy
- ◆ In the preceding section, the statement `m(new Student());`
- ◆ assigns the object `new Student()` to a parameter of the `Object` type. This statement is equivalent to `Object o = new Student(); // Implicit casting`  
`m(o);`  This statement is implicit casting, is legal because an instance of `Student` is automatically an instance of `Object`.

### ❖ Casting from Superclass to Subclass

- ◆ Explicit casting must be used when casting an object from a superclass to a subclass
  - ◆ This type of casting may not always succeed
- ```
Apple x = (Apple)fruit;
Orange x = (Orange)fruit;
```

16



## The Operator instanceof

Use the instanceof operator to test whether an object is an instance of a class:

```
Object myObject = new Circle();
// Some lines of code
/** Perform casting if myObject is an instance of Circle */
if (myObject instanceof Circle) {
    System.out.println("The circle diameter is " +
        ((Circle)myObject).getDiameter());
// Some lines of code
}
```

To help understand casting, you may also consider the analogy of fruit, apple, and orange with the Fruit class as the superclass for Apple and Orange. An apple is a fruit, so you can always safely assign an instance of Apple to a variable for Fruit. However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of Fruit to a variable of Apple.

17

## Casting Demo using Polymorphism

```
1. public class CastingDemo {
2.     /** Main method */
3.     public static void main(String[] args) {
4.         // Declare and initialize two objects
5.         Object object1 = new Circle4(1);
6.         Object object2 = new Rectangle1(1, 1);
7.
8.         // Display circle and rectangle
9.         displayObject(object1);
10.        displayObject(object2);
11.    }
12.
13.    /** A method for displaying an object */
14.    public static void displayObject(Object object) {
15.        if (object instanceof Circle4) {
16.            System.out.println("The circle area is " +
17.                ((Circle4)object).getArea());
18.            System.out.println("The circle diameter is " +
19.                ((Circle4)object).getDiameter());
20.        }
21.        else if (object instanceof Rectangle1) {
22.            System.out.println("The rectangle area is " +
23.                ((Rectangle1)object).getArea());
24.        }
25.    }
26. }
```

The circle area is 3.141592653589793  
The circle diameter is 2.0  
The rectangle area is 1.0

18

## The Object Class and Its Methods

Every class in Java is descended from the java.lang.Object class.

If no inheritance is specified when a class is defined, the superclass of the class is Object.

```
public class Circle {  
    ...  
}
```

Equivalent

```
public class Circle extends Object {  
    ...  
}
```

The Object class `toString()` method returns a string representation of the object.

The Object class `equals()` method compares the contents of two objects.

19

## The `toString()` method in Object class

❖ The `toString()` method returns a string representation of the object. The default implementation returns a string consisting of a class name of which the object is an instance, the at sign (`@`), and a unsigned hexadecimal representation of the hash code of the object.

```
Loan loan = new Loan();  
System.out.println(loan.toString());
```

The code displays something like Loan@15037e5. This message is not very helpful or informative. Usually you should override the `toString` method so that it returns a digestible string representation of the object.

20

## The equals() method in Object class

The equals() method compares contents of two objects.

For example, the equals method is overridden in the Circle class.

```
public boolean equals(Object o) {
    if (o instanceof Circle) {
        return radius == ((Circle)o).radius;
    }
    else
        return false;
}
```

Note The == comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references.

The equals method is intended to test whether two objects have the same contents, provided that the method is modified in the defining class of the objects.

21

## The ArrayList Class

You can create an array to store objects, but the array's size is fixed once the array is created.

Java provides the ArrayList class that can be used to store an unlimited number of objects.

Functionally analogous to the StringBuilder and String classes.

| java.util.ArrayList                  |                                                               |
|--------------------------------------|---------------------------------------------------------------|
| +ArrayList()                         | Creates an empty list.                                        |
| +add(o: Object) : void               | Appends a new element o at the end of this list.              |
| +add(index: int, o: Object) : void   | Adds a new element o at the specified index in this list.     |
| +clear(): void                       | Removes all the elements from this list.                      |
| +contains(o: Object): boolean        | Returns true if this list contains the element o.             |
| +get(index: int) : Object            | Returns the element from this list at the specified index.    |
| +indexOf(o: Object) : int            | Returns the index of the first matching element in this list. |
| +isEmpty(): boolean                  | Returns true if this list contains no elements.               |
| +lastIndexOf(o: Object) : int        | Returns the index of the last matching element in this list.  |
| +remove(o: Object): boolean          | Removes the element o from this list.                         |
| +size(): int                         | Returns the number of elements in this list.                  |
| +remove(index: int) : Object         | Removes the element at the specified index.                   |
| +set(index: int, o: Object) : Object | Sets the element at the specified index.                      |

22

## protected and final Modifiers

- ❖ **protected** modifier can be applied on data and methods in a class
  - ◆ protected data or protected methods in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.
  - ◆ Increasing visibility: **private default protected public**
  - ◆ subclass may override a protected method in its superclass and change its visibility to public. However, a subclass cannot weaken the accessibility of a method defined in the superclass. For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.
- ❖ **final** class modifier means cannot be extended:  

```
final class Math {  
    ...  
}
```
- ❖ The final variable is a constant:  

```
final static double PI = 3.14159;
```
- ❖ The final method cannot be overridden by its subclasses

23

## Suggested Videos

- ❖ [Java \(Beginner\) Programming Tutorials by thenewboston](#)
  - ◆ [Java Programming Tutorial - 49 - Inheritance](#)
  - ◆ [Java Programming Tutorial - 55 - Introduction to Polymorphism](#)
  - ◆ [Java Programming Tutorial - 56 - Polymorphic Arguments](#)
  - ◆ [Java Programming Tutorial - 61 - Simple Polymorphic Program](#)
- ❖ <http://www.youtube.com/user/thenewboston>

24