

# Chapter 11 Inheritance and Polymorphism

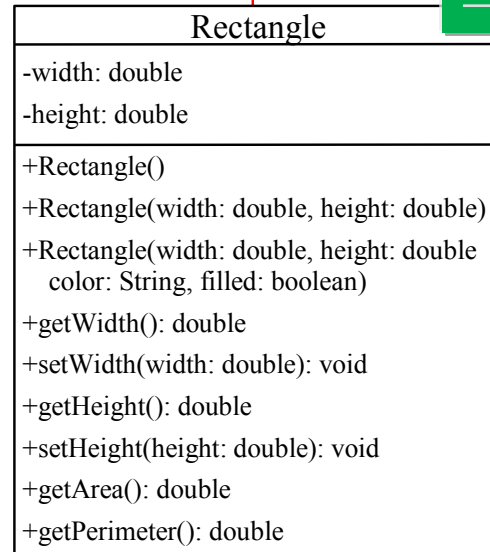
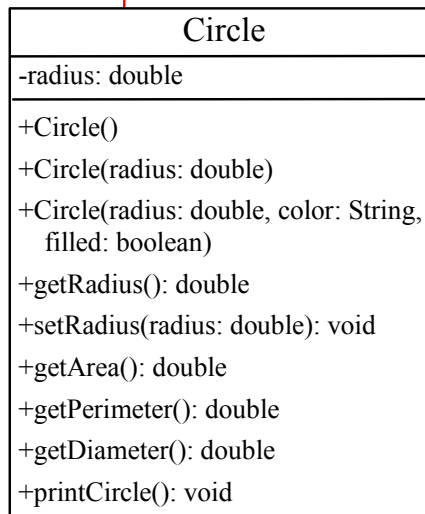
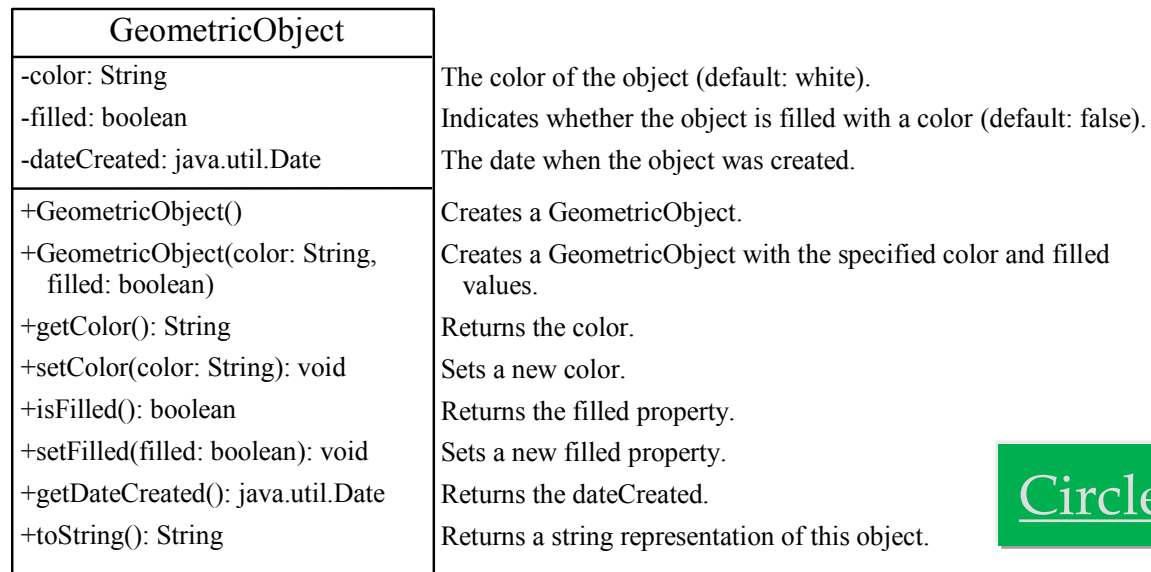
# Motivations

Suppose you will define classes to model circles, rectangles, and triangles. These classes have many common features. What is the best way to design these classes so to avoid redundancy? The answer is to use inheritance.

# Objectives

- ◆ To define a subclass from a superclass through inheritance ( § 11.2).
- ◆ To invoke the superclass's constructors and methods using the **super** keyword ( § 11.3).
- ◆ To override instance methods in the subclass ( § 11.4).
- ◆ To distinguish differences between overriding and overloading ( § 11.5).
- ◆ To explore the **toString()** method in the **Object** class ( § 11.6).
- ◆ To discover polymorphism and dynamic binding ( § § 11.7–11.8).
- ◆ To describe casting and explain why explicit downcasting is necessary ( § 11.9).
- ◆ To explore the **equals** method in the **Object** class ( § 11.10).
- ◆ To store, retrieve, and manipulate objects in an **ArrayList** ( § 11.11).
- ◆ To implement a **Stack** class using **ArrayList** ( § 11.12).
- ◆ To enable data and methods in a superclass accessible from subclasses using the **protected** visibility modifier ( § 11.13).
- ◆ To prevent class extending and method overriding using the **final** modifier ( § 11.14).

# Superclasses and Subclasses



GeometricObject

CircleFromSimpleGeometricObject

RectangleFromSimpleGeometricObject

TestCircleRectangle

# Are superclass's Constructor Inherited?

No. They are not inherited.

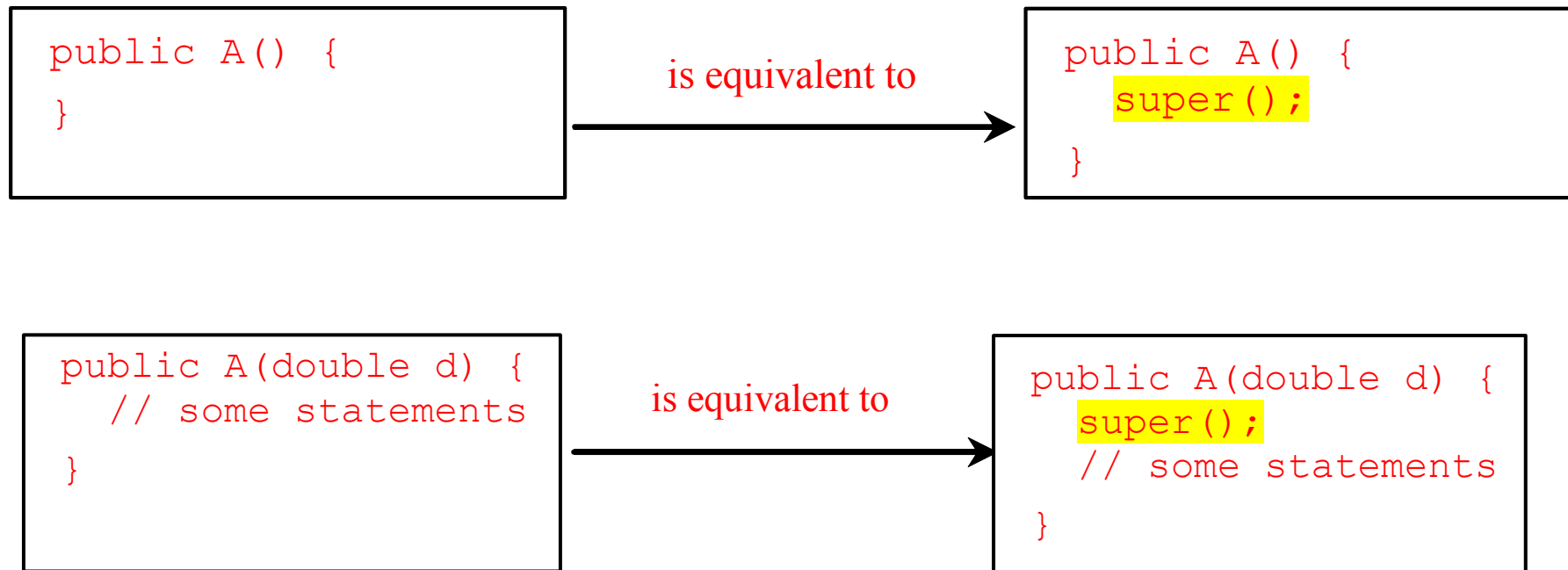
They are invoked explicitly or implicitly.

Explicitly using the super keyword.

A constructor is used to construct an instance of a class. Unlike properties and methods, a superclass's constructors are not inherited in the subclass. They can only be invoked from the subclasses' constructors, using the keyword super. *If the keyword super is not explicitly used, the superclass's no-arg constructor is automatically invoked.*

# Superclass's Constructor Is Always Invoked

A constructor may invoke an overloaded constructor or its superclass's constructor. If none of them is invoked explicitly, the compiler puts super() as the first statement in the constructor. For example,



# Using the Keyword `super`

The keyword `super` refers to the superclass of the class in which `super` appears. This keyword 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 this / super appear first in the constructor.

Java requires that this and super can not co-exist in a constructor.



# Constructor Chaining

Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain. This is known as *constructor chaining*.

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


# Trace Execution

```
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");  
    }  
}
```

1. Start from the  
main method

# Trace Execution

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



# Trace Execution

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

3. Invoke Employee's no-arg constructor

# Trace Execution

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

4. Invoke Employee(String)  
constructor

# Trace Execution

```
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 People {  
    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);  
    }  
}
```

```
public Employee(String s) {  
    System.out.println(s);  
}
```

5. Invoke People() constructor

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

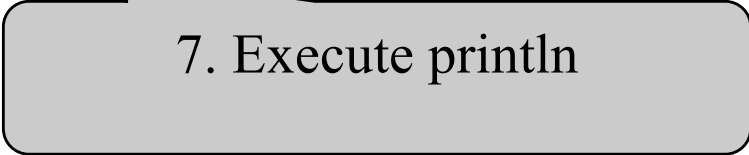
# Trace Execution

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

6. Execute println

# Trace Execution

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

A diagram consisting of a grey rounded rectangle containing the text "7. Execute println". A line originates from the left side of this rectangle and points to the `System.out.println(s);` statement within the `public Employee(String s) {` block of the `Employee` class.

7. Execute println



# Trace Execution

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



8. Execute println

# Trace Execution

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

9. Execute println

# Example on the Impact of a Superclass without no-arg Constructor

Find out the errors in the program:

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

# Defining a Subclass

A subclass inherits from a superclass. You can also:

- ◆ Add new properties
- ◆ Add new methods
- ◆ Override the methods of the superclass

# Calling Superclass Methods

You could rewrite the printCircle() method in the Circle class as follows:

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

# Overriding Methods in the Superclass

A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as *method overriding*.

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

# NOTE

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.

# NOTE

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.



# Overriding vs. 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 overrides the method in B  
    public void p(double i) {  
        System.out.println(i);  
    }  
}
```

```
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);  
    }  
}
```

# 12 Rules of Overriding in Java You Should Know

- ◆ <http://www.codejava.net/java-core/the-java-language/12-rules-of-overriding-in-java-you-should-know>

12 Rules of Overriding in Java You Should Know.pdf

RTpolyTest

# 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 toString() method in Object

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 number representing this 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.

# Polymorphism

Polymorphism means that a variable of a supertype can refer to a subtype object.

A class defines a type. A type defined by a subclass is called a *subtype*, and a type defined by its superclass is called a *supertype*.

**Circle** is a subtype of **GeometricObject** and **GeometricObject** is a supertype for **Circle**.

[PolymorphismDemo](#)

# Polymorphism, Dynamic Binding and Generic Programming

```
public class PolymorphismDemo {  
    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.

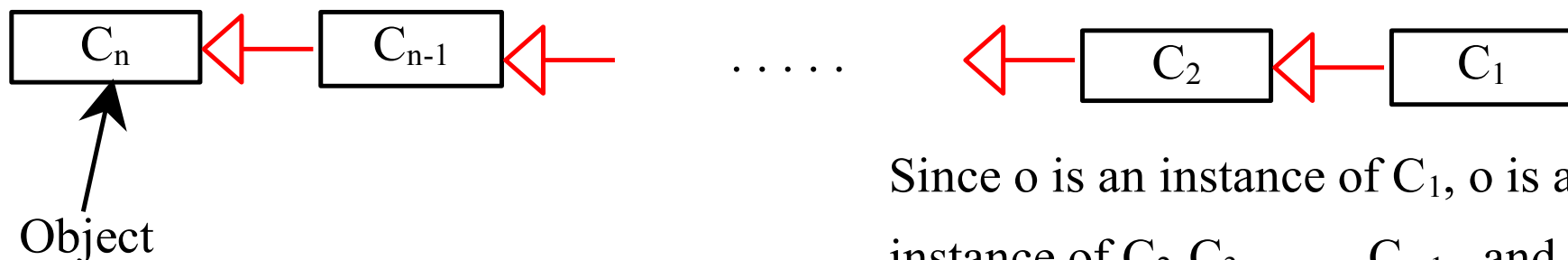
An object of a subtype can be used wherever its supertype value is required. This feature is known as *polymorphism*.

When the 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*.

DynamicBindingDemo

# Dynamic Binding

Dynamic binding works as follows: Suppose an object  $o$  is an instance of classes  $C_1, C_2, \dots, C_{n-1}$ , and  $C_n$ , where  $C_1$  is a subclass of  $C_2$ ,  $C_2$  is a subclass of  $C_3$ , ..., and  $C_{n-1}$  is a subclass of  $C_n$ . That is,  $C_n$  is the most general class, and  $C_1$  is the most specific class. In Java,  $C_n$  is the Object class. If  $o$  invokes a method  $p$ , the JVM searches the implementation for the method  $p$  in  $C_1, C_2, \dots, C_{n-1}$  and  $C_n$ , in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.



Since  $o$  is an instance of  $C_1$ ,  $o$  is also an instance of  $C_2, C_3, \dots, C_{n-1}$ , and  $C_n$

# Method Matching vs. Binding

Matching a method signature and binding a method implementation are two issues. The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time.

A method may be implemented in several subclasses. The Java Virtual Machine dynamically binds the implementation of the method at runtime.



# Generic Programming

```
public class PolymorphismDemo {
    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. This is known as 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.


# Casting Objects

We have already used the casting operator to convert variables of one primitive type to another. *Casting* can also be used to convert an object of one class type to another within an inheritance hierarchy.

```
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);
```



The statement `Object o = new Student()`, known as implicit casting, is legal because an instance of `Student` is automatically an instance of `Object`.

# Why Casting Is Necessary?

Suppose you want to assign the object reference `o` to a variable of the `Student` type using the following statement:

```
Student b = o;
```

A compile error would occur. Why does the statement **Object o = new Student()** work and the statement **Student b = o** doesn't? This is because a `Student` object is always an instance of `Object`, but an `Object` is not necessarily an instance of `Student`. Even though you can see that `o` is really a `Student` object, the compiler is not so clever to know it. To tell the compiler that `o` is a `Student` object, use an explicit casting. The syntax is similar to the one used for casting among primitive data types. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```
Student b = (Student)o; // Explicit casting
```

# 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;
```

# The instanceof Operator

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());  
    ...  
}
```

ClassAndInstance

# Example: Demonstrating Polymorphism and Casting

This example creates two geometric objects: a circle, and a rectangle, invokes the `displayGeometricObject` method to display the objects. The `displayGeometricObject` displays the area and diameter if the object is a circle, and displays area if the object is a rectangle.

[CastingDemo](#)

[UpMonkey](#)

# The equals Method

The `equals()` method compares the contents of two objects. The default implementation of the `equals` method in the `Object` class is as follows:

```
public boolean equals(Object obj) {  
    return this == obj;  
}
```

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;  
}
```

`equalstest.EqualsTest`

# 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. The `==` operator is stronger than the `equals` method, in that the `==` operator checks whether the two reference variables refer to the same object.



# The protected Modifier

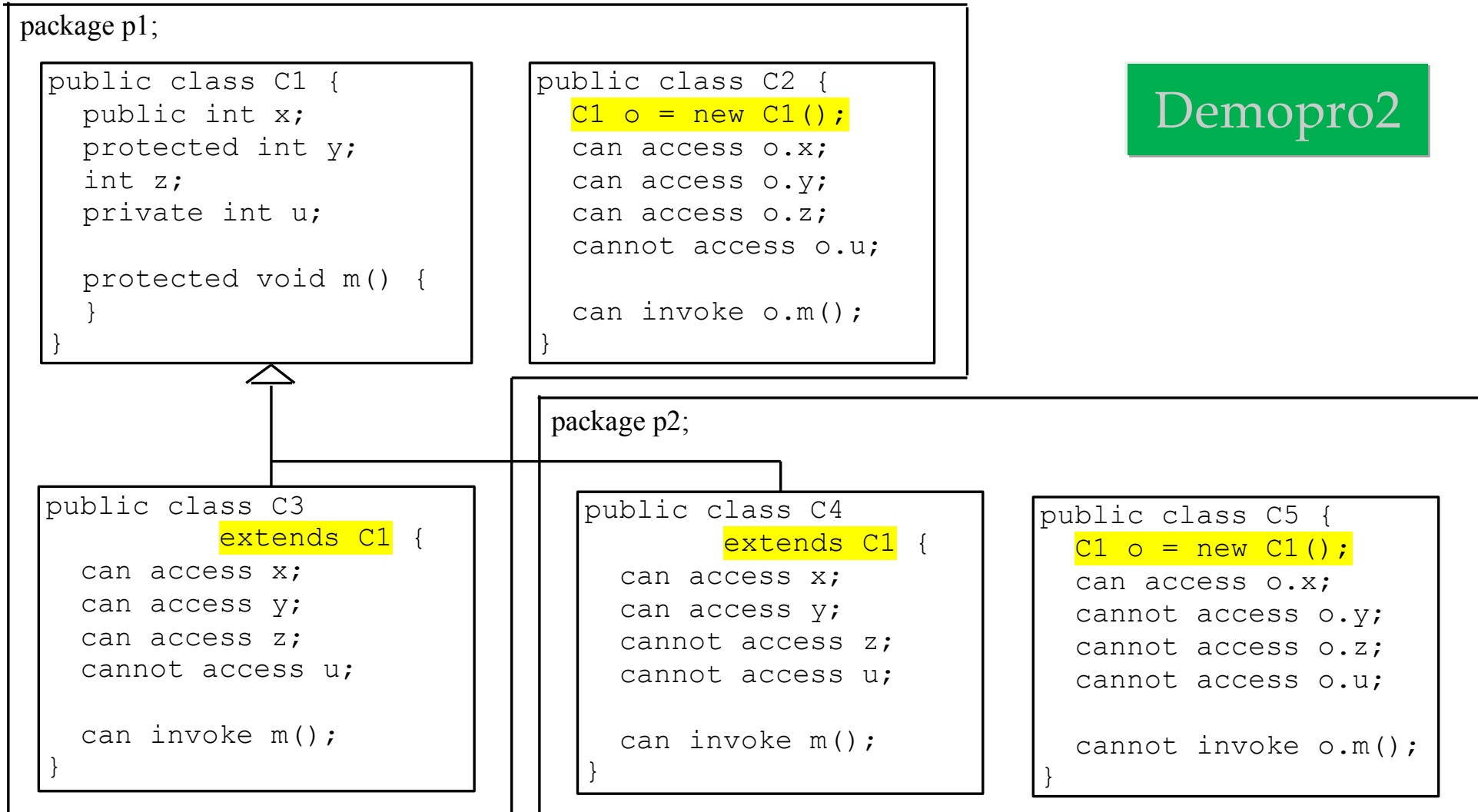
- ◆ The `protected` modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.
- ◆ `private`, `default`, `protected`, `public`

Visibility increases  
—————→  
`private`, `none` (if no modifier is used), `protected`, `public`

# Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

# Visibility Modifiers



# A Subclass Cannot Weaken the Accessibility

A 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.*

# NOTE

The modifiers are used on classes and class members (data and methods), except that the final modifier can also be used on local variables in a method. A final local variable is a constant inside a method.

# The `final` Modifier

- ◆ The `final` class 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.

```
final int getLength(String s) {  
    ...  
}
```

# Note

- ♦ final vs. abstract

- should not be used to modify a class or method at the same time

- ♦ final vs. immutable

- final refers to references while immutable refers to objects. Assigning the final modifier to a reference means it cannot change to point to another object, but the object itself can be modified if it is mutable

# Design Hints for Inheritance

1. Place common operations and fields in the superclass.
2. Don't use protected fields.
3. Use inheritance to model the “is—a” relationship.
4. Don't use inheritance unless all inherited methods make sense.
5. Don't change the expected behavior when you override a method.
6. Use polymorphism, not type information.
7. Don't overuse reflection.\*

Refer to Core Java Vol 1