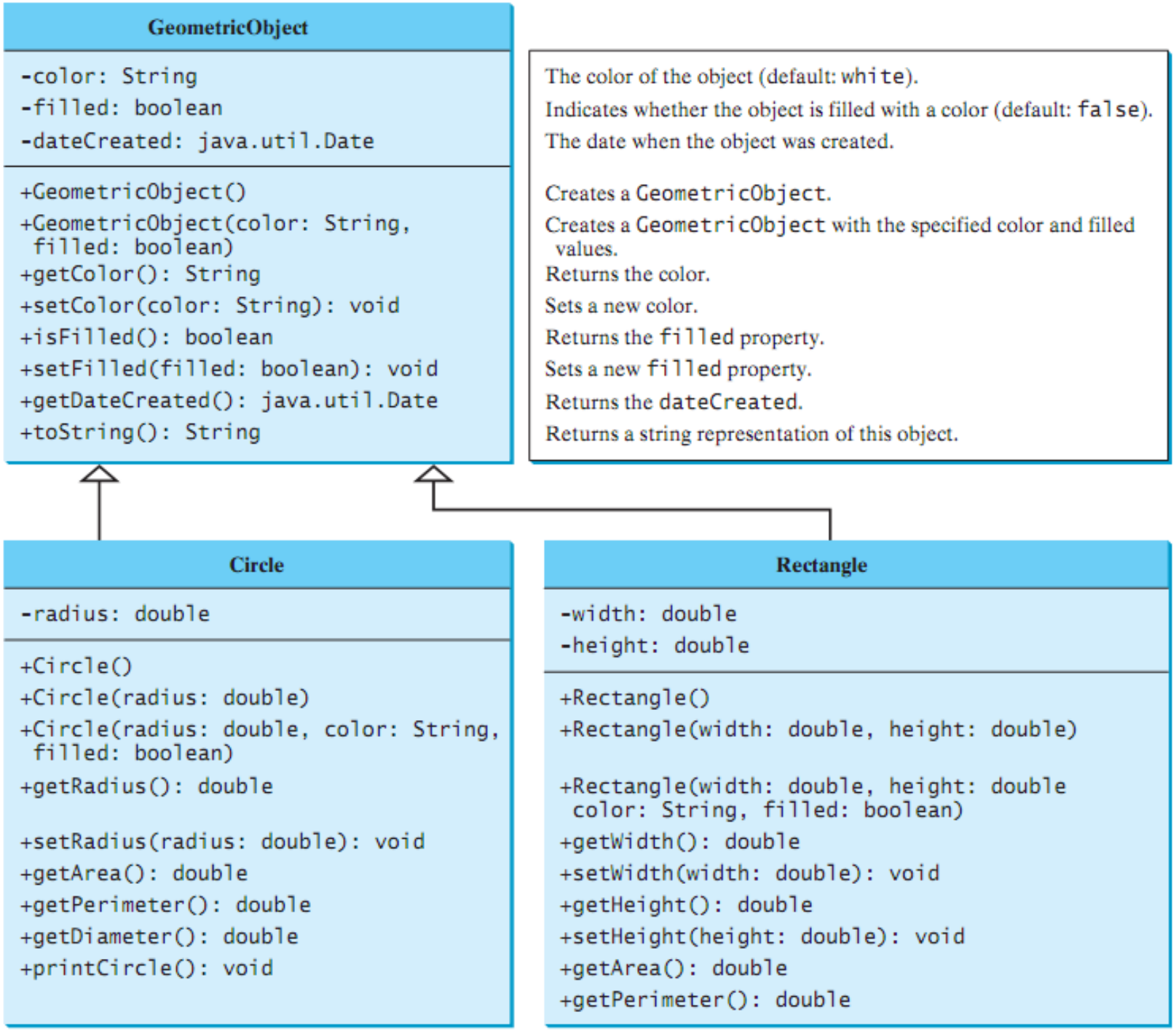
Chapter 11 Inheritance and Polymorphism

为什么需要继承性？

- 设想你编写了这么几个类：circles, rectangles, triangles，显然这些类有很多共性（如颜色），因此会有很多代码是重复的。
- 为了更好地组织这些类，我们可以把这些类的公共部分先做成一个类GeometricObject，然后circles, rectangles, triangles就在这个类的基础上扩充，这样代码就没有冗余了。
- 对一个已有的类进行扩充，你只需要把原有的类继承下来，不需要改动已有的那个类。

超类（Superclass）和子类（Subclasses）

- 对一个已有的类做扩充，叫做继承。被继承的那个已有类叫做超类（superclass），新的那个类叫子类（subclass）。
- 继承性是面向对象的第二个特性。有些面向对象的语言把超类叫做父类，把子类叫做派生类，意思是一样的。
- 继续上面那个例子，先画UML图：


GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
+GeometricObject()
+GeometricObject(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
The color of the object (default: white).
Indicates whether the object is filled with a color (default: false).
The date when the object was created.
Creates a GeometricObject.
Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns the dateCreated.
Returns a string representation of this object.
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: String, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double
+printCircle(): void
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

SimpleGeometricObject的部分代码

```
public class SimpleGeometricObject {
 private String color = "white";
 private boolean filled;
 private java.util.Date dateCreated;
public SimpleGeometricObject() {
 dateCreated = new java.util.Date();
}
public SimpleGeometricObject(String color, boolean filled) {
 this.color = color;    this.filled = filled;
 }
…
```

CircleFromSimpleGeometricObject的部分代码

```
public class CircleFromSimpleGeometricObject extends
SimpleGeometricObject {
 private double radius;
 public CircleFromSimpleGeometricObject() {
 }
 public CircleFromSimpleGeometricObject(double radius) {
 this.radius = radius;
 }
 public CircleFromSimpleGeometricObject(double radius,
 String color, boolean filled) {
…
```

测试程序

LISTING 11.4 TestCircleRectangle.java

```
1  public class TestCircleRectangle {
2    public static void main(String[] args) {
3       CircleFromSimpleGeometricObject circle =
4         new CircleFromSimpleGeometricObject(1);                    Circle object
5      System.out.println("A circle " + circle.toString());          invoke toString
6      System.out.println("The color is " + circle.getColor());      invoke getColor
7      System.out.println("The radius is " + circle.getRadius());
8      System.out.println("The area is " + circle.getArea());
9      System.out.println("The diameter is " + circle.getDiameter());
10
11     RectangleFromSimpleGeometricObject rectangle =
12       new RectangleFromSimpleGeometricObject(2, 4);                Rectangle object
13     System.out.println("\nA rectangle " + rectangle.toString());   invoke toString
14     System.out.println("The area is " + rectangle.getArea());
15     System.out.println("The perimeter is " +
16       rectangle.getPerimeter());
17   }
18 }
```

如何继承

- 继承的语法很简单，如下：

```
public class CircleFromSimpleGeometricObject
    extends SimpleGeometricObject
```

CircleFromSimpleGeometricObject就是子类，SimpleGeometricObject就是超类。

注意Java只允许有一个父类，所以extends后面有且只有一个类名。

关于超类的构造方法

- 当继承发生的时候，超类的public成员会被子类自动继承。考虑到超类的构造方法一般也是public，那么这个方法会被继承下来吗？
- 答案是：不会。不过它们会在子类中被自动调用，也可以在子类中使用super关键字直接调用。
- super和this用法类似，例如super( )表示调用超类的无参构造方法，super(arg1, arg2)表示调用超类有两个参数的构造方法。

超类的构造方法会被自动调用

在构造方法中，可以调用(1)重载的另一个构造方法(2)超类的构造方法。如果两种情况都没有发生，那么编译器会自动在构造方法插入 super(); 作为第一个语句，用来强制调用父类构造方法。例如：

```
public A() {
}
```

is equivalent to

```
public A() {
  super();
}
```

```
public A(double d) {
  // some statements
}
```

is equivalent to

```
public A(double d) {
  super();
  // some statements
}
```

super

Super关键字用来表示超类，这个关键字一般有两种用途：

- 调用超类构造方法，如前面的例子；
- 调用超类的其它方法，这个后面会遇到。

需要提醒的是，使用super调用超类的构造方法，只能出现在子类的构造方法的第一个语句。

构造方法链的例子，注意以下三个类之间的关系

```
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");
  }
}
```

单步执行一下

```
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. 从main开始运行

单步执行一下

```
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");
  }
}
```

2. 调用Faculty的无参构造方法

单步执行一下

```
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");
  }
}
```

3. 调用Employee的无参构造方法

单步执行一下

```
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");
  }
}
```

4. 调用Employee(String)构造方法

请思考：为何这一步不是调用Person()构造方法，而是调用Employee(String)构造方法

单步执行一下

```
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");
  }
}
```

5. 调用Person()构造方法

单步执行一下

```
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");
  }
}
```

6. 执行println

单步执行一下

```
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");
  }
}
```

7. 执行println

单步执行一下

```
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");
  }
}
```

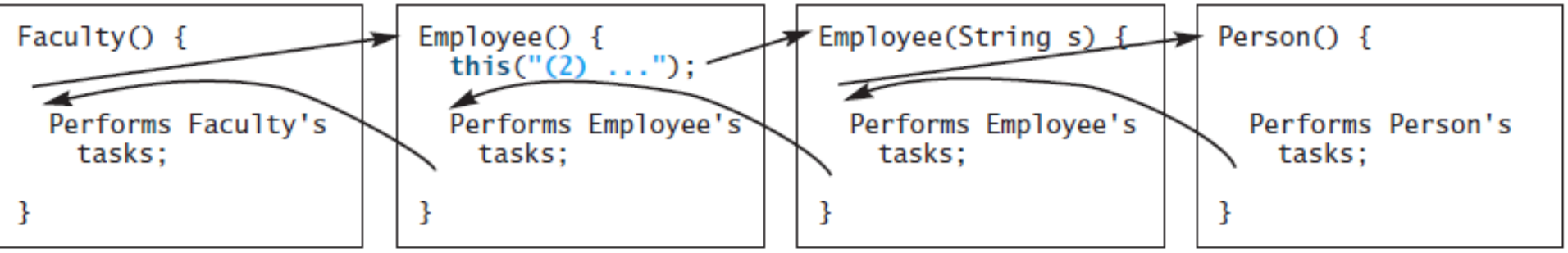
8. 执行println

单步执行一下

```
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");
  }
}
```

9. 执行println

总结一下完整构造链



- 要点就是：1. 父类的构造方法总是需要被调用的，无论显式还是隐式；2. 如果子类构造方法中，首先调用了自己的另一个构造方法M，那么此处可以忽略父类构造方法的调用，但是在M中还是要重复上述步骤1,2。

当超类没有提供无参构造方法的时候...

找出下列程序的错误：

```
public class Apple extends Fruit {
}

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

经验教训就是：给每一个类提供一个无参的构造方法是一个好习惯。

子类可以做什么

子类继承了超类的属性和方法，你可以在此基础上继续做：

- 添加新的属性
- 添加新的方法
- 覆盖超类的方法

覆盖超类的方法

子类继承了超类的方法，但是有时候需要对某个方法进行修改以适应子类的需求。这种情况下可以采用覆盖机制，在子类中定义一个和父类完全相同的方法，达到修改父类方法的目的，这种技术就是所谓的**方法覆盖**（*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;
  }
}
```

注意

- 不是每一个超类的方法都能被覆盖，例如被**private**修饰的超类方法就不行。当然子类中依然可以定义和超类private一样的方法，不过这个不叫覆盖，因为这两个方法是完全没有关联的。
- 同样的，使用**static**修饰的超类方法也不能被子类覆盖，子类虽然依然可以定义一个完全相同的方法，但是只能达到隐藏父类方法的目的，达不到覆盖的效果。

覆盖Overriding vs. 重载Overloading

- 下面代码中，左侧是覆盖，右侧是重载，看出区别来了吗？

```
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);
  }
}
```

运行结果

- 上个例子中，左侧的a.p(10) 和 a.p(10.0) 都调用A类的同一个方法 p(double i)，输出结果都是10.0；右侧的a.p(10)调用了A中的 p(int i)，输出结果10，a.p(10.0)调用了B中的p(double i)，输出结果20.0。

解释

- 覆盖发生在具有继承关系的两个类中。
- 重载可以发生在同一个类中，也可以发生在具有继承关系的两个类中。
- 覆盖方法具有完全一样的方法签名（方法名和参数表）以及返回类型。
- 重载方法只是方法名一样，但是参数表肯定不同。

@Override

- 由于覆盖方法要求很严，所以可以让Java检查是否覆盖成功，这就是**@Override**的作用。它的用法很简单，放置在需检查的子类方法之前即可：

```
public class CircleFromSimpleGeometricObject
    extends SimpleGeometricObject {
 @Override //提示检查覆盖，如果失败，编译报错
 public String toString() {
 return super.toString() + "\nradius is " + radius;
 }
}
```

Object类

- 其实任何一个Java类都有超类，因为如果一个类没有extends一个类，Java会默认extends java.lang.Object。换句话说， Object这个类是Java中所有类的老祖宗。

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

Equivalent

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

Object的toString()方法

- Object预先定义了一个toString()方法，用来输出一个类本身的描述性信息，其原型如下：
  public String toString();
- 由于这个toString会被子类所继承，所以任何一个类都可以调用这个方法，例如：

  Loan loan = new Loan();

  System.out.println(loan.toString());
- 上面这个输出会得到类似**Loan@15037e5**这样的信息，不过显然这个结果没什么用处。所以通常每一个类都会很自觉地覆盖toString方法以输出更有意义的信息。

多态性和动态绑定

```
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";
  }
}
```

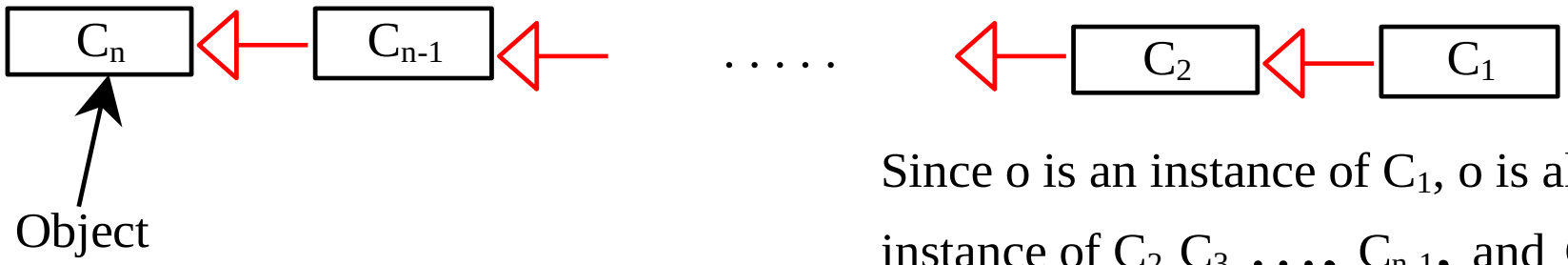
当需要某种类型时，可以由它的子类型代替，这就是所谓的多态性（*polymorphism*）。例如左边的m方法，参数需要Object类型，所以可以传递任何Object类型的子类型给它。

当方法m被调用时，对象x的toString方法会被调用。由于x可能是GraduateStudent, Student, Person, 或者Object，这几个类都有自己的toString方法，所以JVM会在运行的时候，依据传入的参数类型，自动决定运行哪个版本的toString，这个技术叫做动态绑定（*dynamic binding*）。

动态绑定（Dynamic Binding）

- 动态绑定的运行机制如下：

假设对象o 是这几个类的实例 $C_1, C_2, ..., C_{n-1}, C_n$, 并且 C_1 是 C_2的子类, C_2 是 C_3的子类, ..., C_{n-1} 是C_n的子类。也就是说 C_n 是最普通的类，C_1 则是最特殊的类。在Java中，C_n 就是Object类。 如果 o 调用了一个方法p，JVM将沿着 $C_1, C_2, ..., C_{n-1}, C_n$的顺序搜索方法p的具体实现，直到找到方法p的第一个实现为止。当然，方法p是一定能找到的，如果找不到的话编译就报错了，轮不到运行出错 。



继续解释上面那个例子

```
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";
  }
}
```

```
Student
Student
Person
java.lang.Object@130c19b
```

注意这几个类的继承顺序是：

Object←Person←Student←GraduateStudent，

依据动态绑定的机制，不难理解这个输出。

方法匹配 vs. 绑定

- 方法匹配和方法绑定是两个不同的问题。
- 方法匹配指的是编译器通过方法的参数类型，参数顺序来匹配同名的方法，这是在编译阶段完成的事情。
- 方法绑定指的是当某个方法在众多子类中都实现了，由于函数签名完全相同，编译器无法在编译阶段决定该调用哪一个方法，只能延迟到运行阶段，由JVM根据传入的参数类型，动态绑定到某个方法上。

对象类型转换

- 对象的类型是可以转换的，分为强制转换和隐式转换两种。下面是一个例子：

```
Object o = new Student();
m(o);
```

- 上面是一个隐式转换。这个转换是完全合法的。因为根据继承关系，Student类是Object类的子类，而在Java中，**子类的实例自动是超类的实例。**

为什么需要类型转换

继续上面那个例子，我们把o转换回来：

```
Object o = new Student();
m(o);
Student b = o;
```

很不幸，第3行编译器报错了。虽然你也知道这个o明明就是一个Student，可是Java不相信。这个时候，你只能借助显式的强制类型转换才行：

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

从超类到子类的转换

显式转换用于将超类强制转换为子类（反之是自动转的，不用强制）。下面是另两个例子。友情提醒一下，这种转换并不总是成功，有风险的。

```
Apple x = (Apple)fruit;

Orange x = (Orange)fruit;
```

Instanceof运算符

Instanceof运算符用于测试一个对象是否是某个类的实例。建议在对象强制转换之前进行该测试，可以确保转换的安全。下面是一个例子：

```
Object myObject = new Circle();
... // 其它无关代码
/** 测试myObject是否Circle类的实例 */
if (myObject instanceof Circle) {
  System.out.println("The diameter is " +
  ((Circle)myObject).getDiameter());
  ... // 其它无关代码
}
```

多态性和类型转换的例子

类CircleFromSimpleGeometricObject表示一个圆，类RectangleFromSimpleGeometricObject表示一个矩形。阅读下面这个例子。

```java
public class CastingDemo {
  /** Main method */
  public static void main(String[] args) {
    // Create and initialize two objects
    Object object1 = new CircleFromSimpleGeometricObject(1);
    Object object2 = new RectangleFromSimpleGeometricObject(1, 1);

    // Display circle and rectangle
    displayObject(object1);
    displayObject(object2);
  }

  /** A method for displaying an object */
  public static void displayObject(Object object) {
    if (object instanceof CircleFromSimpleGeometricObject) {
      System.out.println("The circle area is " +
        ((CircleFromSimpleGeometricObject)object).getArea());
      System.out.println("The circle diameter is " +
        ((CircleFromSimpleGeometricObject)object).getDiameter());
    }
    else if (object instanceof
                  RectangleFromSimpleGeometricObject) {
      System.out.println("The rectangle area is " +
        ((RectangleFromSimpleGeometricObject)object).getArea());
    }
  }
}
```

Equals方法

equals()方法用于比较两个对象的内容是否完全相同（注意是**内容比较**，不要求是同一个对象）。这个方法在Object类已经被定义，但是一般的类都会覆盖这个方法以便进行更准确的比较。下面是一个例子：

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

关于==运算符

- ==运算符在比较基本数据类型时，比较的是两个值是否相等。例如：
  int i=1, j=1; if (i == j) printf(“值相等”);
- ==运算符在比较对象类型时，比较的是两个对象是否引用了相同的实例（即指向同一个内存地址）。
- equals()仅仅比较内容是否相同，所以如果两个对象==成立，显然equals()一定成立；反之就不一定了。

对象数组类ArrayList

- 如果有一堆的对象要存储，例如学生对象，很显然要用到数组。尽管Java的数组确实是动态的，可以在运行时再指定大小，但是数组大小一旦指定，就没有机会再次调整大小。
- 设想你打算处理一门课程的选课学生信息，这时候会面临一个问题：学生数是变动的。有人会退课，有人会补进来，因此使用数组无法解决这个问题。
- Java提供了**java.util.ArrayList**类专门用来处理这种情况，它是一个真正的动态对象数组类，可以根据需要自动调整数组大小。

ArrayList的定义

- ArrayList是这样定义的：
  - java.util.ArrayList<E>

那个<E>的写法看上去有些奇怪，它表示泛型（generic type），意思是**不针对特定类型，任何类型都可以**。所以ArrayList可以做成任何类型的动态数组。不过你在使用之前必须指定类型，以便它分配内存。例如把ArrayList用作一个字符串数组：

```
ArrayList<String> cities = new ArrayList<String>();
```

需要注意的是ArrayList要求**元素是对象**才行，所以下面是错误的用法：

```
ArrayList<int> error = new ArrayList<int>();
```

java.util.ArrayList的方法简介

java.util.ArrayList<E>	
+ArrayList()	Creates an empty list.
+add(o: E): void	Appends a new element o at the end of this list.
+add(index: int, o: E): 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): E	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): boolean	Removes the element at the specified index.
+set(index: int, o: E): E	Sets the element at the specified index.

一个例子—1/3

```
import java.util.ArrayList;

public class TestArrayList {
public static void main(String[] args) {
// Create a list to store cities
ArrayList<String> cityList = new ArrayList<String>();
// Add some cities in the list
cityList.add("London");
// cityList now contains [London]
cityList.add("Denver");
// cityList now contains [London, Denver]
cityList.add("Paris");
// cityList now contains [London, Denver, Paris]
cityList.add("Miami");
// cityList now contains [London, Denver, Paris, Miami]
cityList.add("Seoul");
// Contains [London, Denver, Paris, Miami, Seoul]
cityList.add("Tokyo");
// Contains [London, Denver, Paris, Miami, Seoul, Tokyo]
```

一个例子—2/3

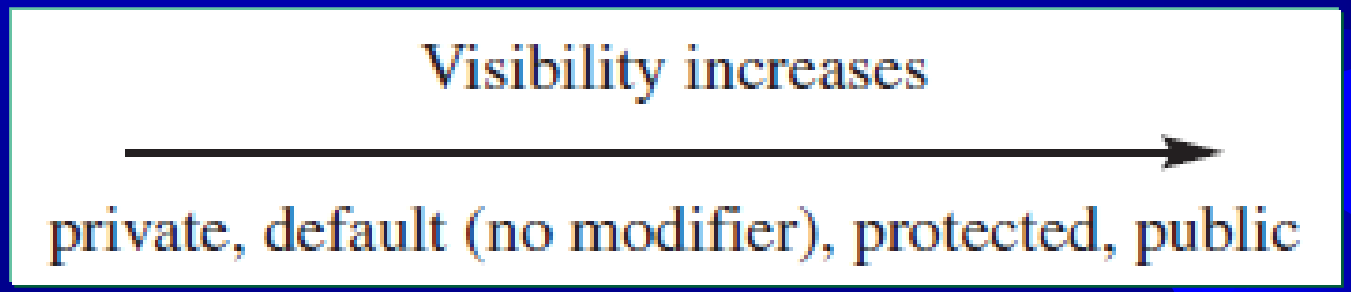
```
System.out.println("List size? " + cityList.size());
//输出: List size? 6

System.out.println("Is Miami in the list? "
+ cityList.contains("Miami"));
//输出: Is Miami in the list? true

System.out.println("The location of Denver in the list? "
+ cityList.indexOf("Denver"));
//输出: The location of Denver in the list? 1

System.out.println("Is the list empty? " + cityList.isEmpty());
//输出: Is the list empty? false

// Insert a new city at index 2
cityList.add(2, "Xian");
// Contains [London, Denver, Xian, Paris, Miami, Seoul, Tokyo]
```

一个例子—3/3

```
// Remove a city from the list
cityList.remove("Miami");
// Contains [London, Denver, Xian, Paris, Seoul, Tokyo]

// Remove a city at index 1
cityList.remove(1);
// Contains [London, Xian, Paris, Seoul, Tokyo]

// Display the contents in the list
System.out.println(cityList.toString());
//输出: [London, Xian, Paris, Seoul, Tokyo]

// Display the contents in the list in reverse order
for (int i = cityList.size() - 1; i >= 0; i--)
System.out.print(cityList.get(i) + " ");
}
//输出: Tokyo Seoul Paris Xian London

}
```

protected修饰符

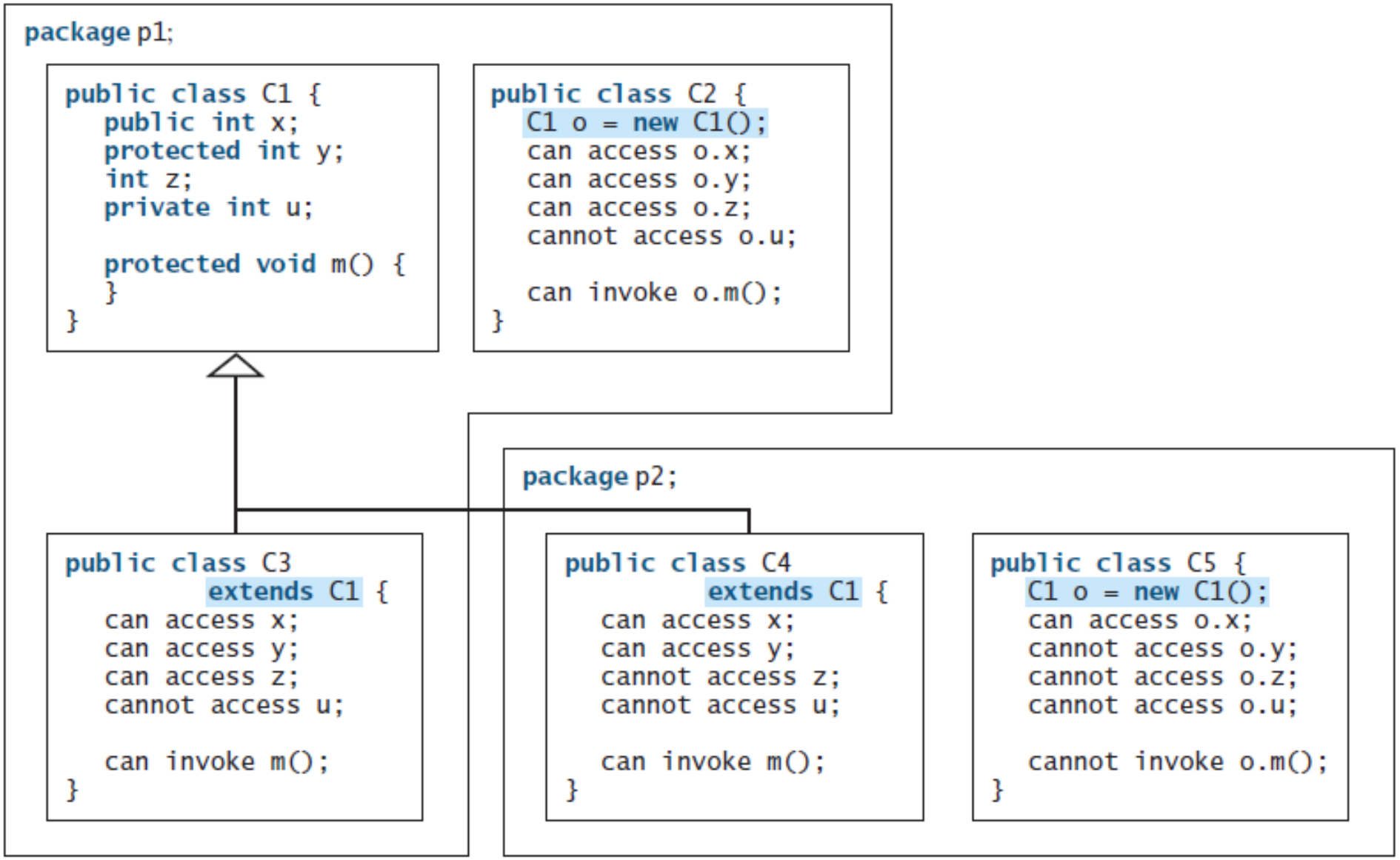
- protected修饰符可以用在类中的属性和方法上。一个public class的protected的成员可以被同一个包中的任意类或其子类所访问，即使子类位于不同的包。
- 对于四种可见性修饰符private, default, protected, public，其开放程度排序如下：



存取权限小结

Modifier on members in a class	*Accessed from the same class*	*Accessed from the same package*	*Accessed from a subclass in a different package*	*Accessed from a different package*
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default (no modifier)	✓	✓	–	–
private	✓	–	–	–

可见性修饰符举例


package p1;
public class C1 {
public int x;
protected int y;
int z;
private int u;
protected void m() {
}
}
public class C2 {
C1 o = new C1();
can access o.x;
can access o.y;
can access o.z;
cannot access o.u;
can invoke o.m();
}
package p2;
public class C3
extends C1 {
can access x;
can access y;
can access z;
cannot access u;
can invoke m();
}
public class C4
extends C1 {
can access x;
can access y;
cannot access z;
cannot access u;
can invoke m();
}
public class C5 {
C1 o = new C1();
can access o.x;
cannot access o.y;
cannot access o.z;
cannot access o.u;
cannot invoke o.m();
}

存取权限的继承限制

- 子类可以覆盖超类的protected方法，并将其可见性提升为public。
- 不过，子类不能降低超类的存取权限，例如超类中定义为public的方法，子类覆盖此方法后，也必须将其定义为public，而不能降低为protected。

Final修饰符

- 以final修饰的类是不能被继承的，例如：

```
final class Math {
  ...
 }
```

- 以final修饰的变量是常量，例如：

```
final static double PI = 3.14159;
```

- 以final修饰的方法是不能被子类覆盖的。

THE END