11 Inheritance and Polymorphism

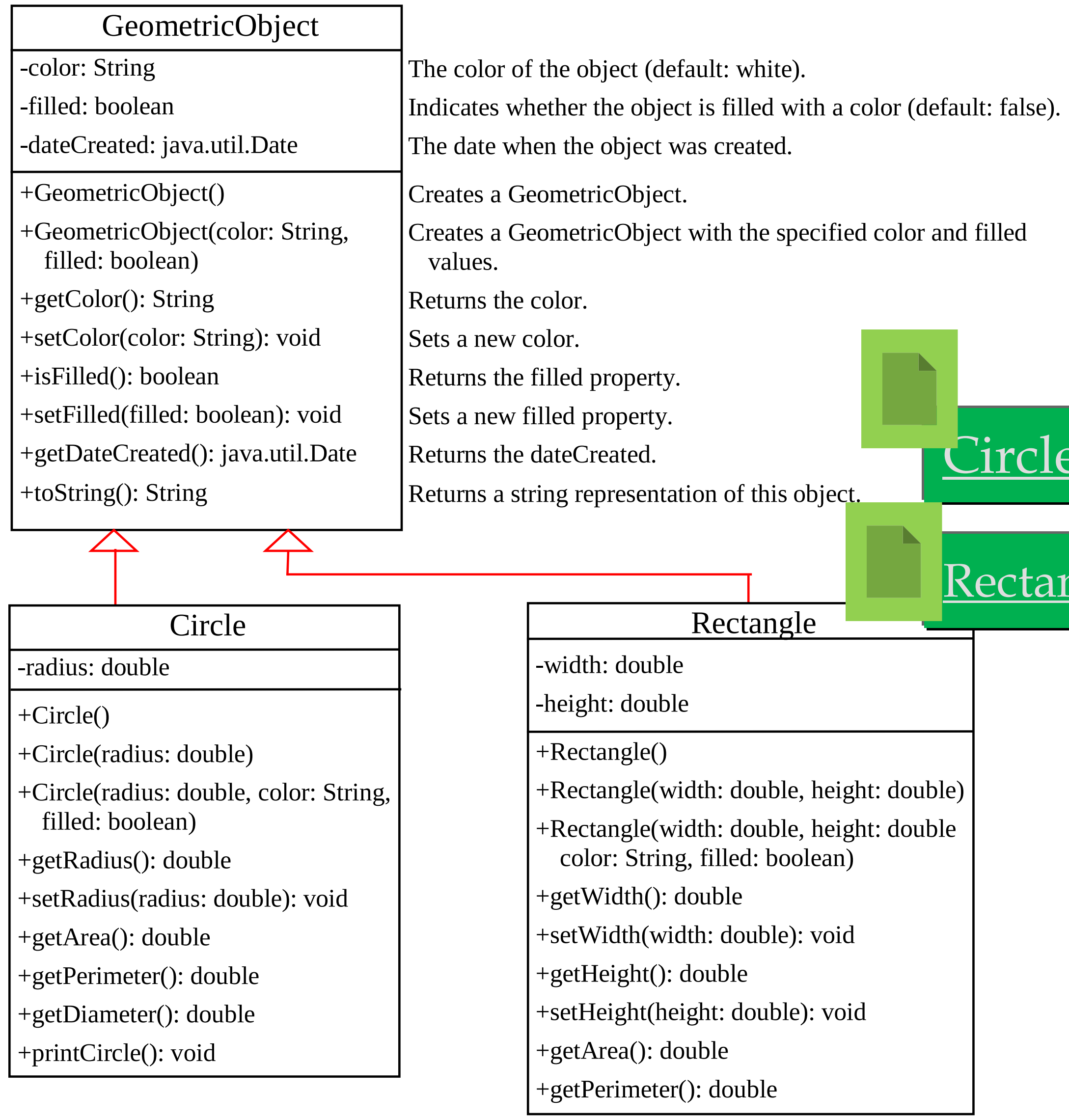
Motivations

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
-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
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

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.

GeometricObject

CircleFromSimpleGeometricObject

RectangleFromSimpleGeometricObject

TestCircleRectangle

Run

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,

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

is equivalent to

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

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

is equivalent to

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

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 super appear first in the 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 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");
  }
}
```

animation

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

animation

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");
  }
}
```

2. Invoke Faculty constructor

animation

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");
  }
}
```

3. Invoke Employee’s no-arg constructor

animation

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");
  }
}
```

4. Invoke Employee(String) constructor

animation

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");
  }
}
```

5. Invoke Person() constructor

animation

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");
  }
}
```

6. Execute println

animation

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");
  }
}
```

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 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. Execute println

animation

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");
  }
}
```

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*.

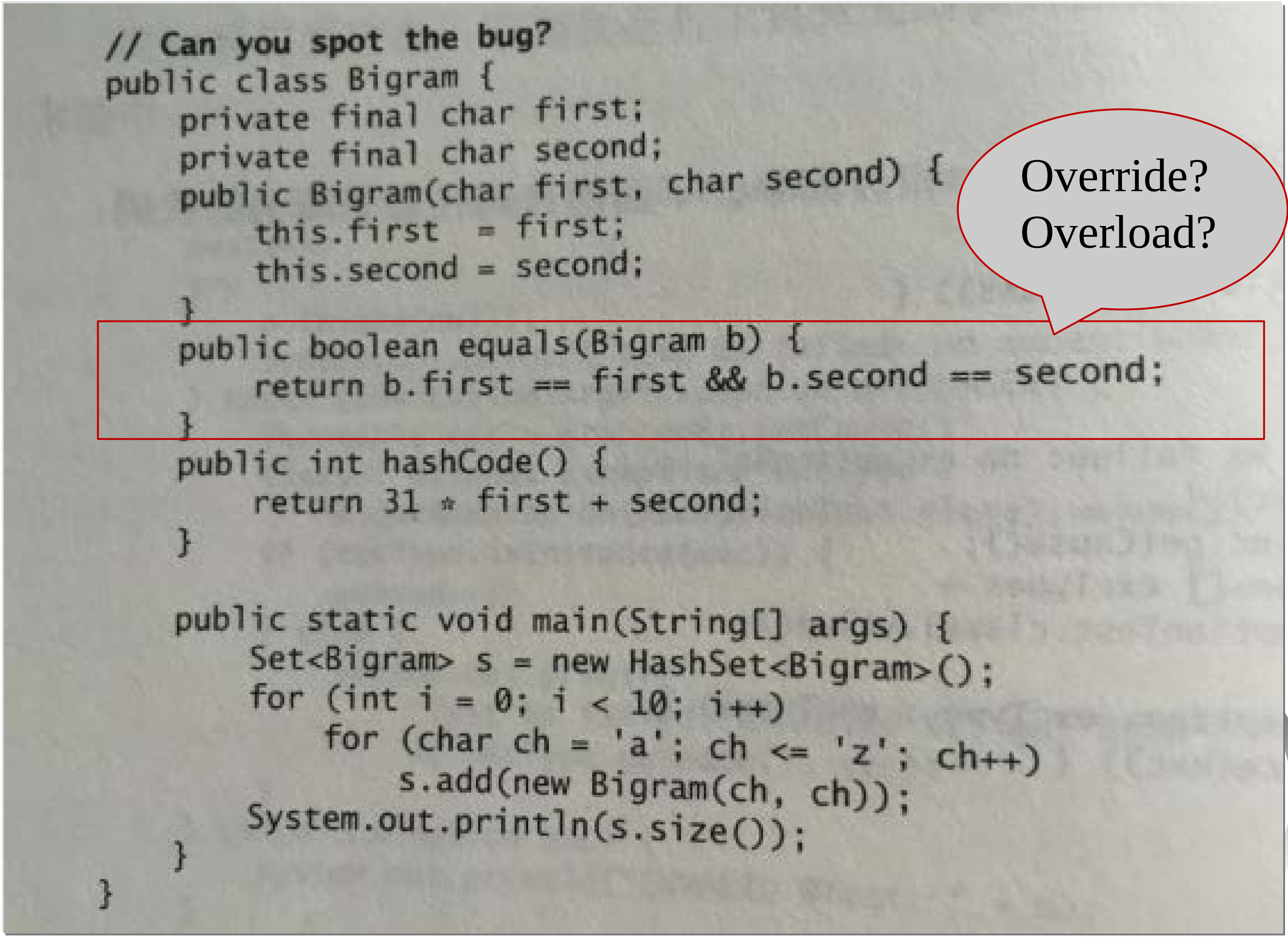
```
public class Circle extends GeometricObject {
  // Other methods are omitted


  /** 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.



```java
// Can you spot the bug?
public class Bigram {
    private final char first;
    private final char second;
    public Bigram(char first, char second) {
        this.first  = first;
        this.second = second;
    }
    public boolean equals(Bigram b) {
        return b.first == first && b.second == second;
    }
    public int hashCode() {
        return 31 * first + second;
    }

    public static void main(String[] args) {
        Set<Bigram> s = new HashSet<Bigram>();
        for (int i = 0; i < 10; i++)
            for (char ch = 'a'; ch <= 'z'; ch++)
                s.add(new Bigram(ch, ch));
        System.out.println(s.size());
    }
}
```

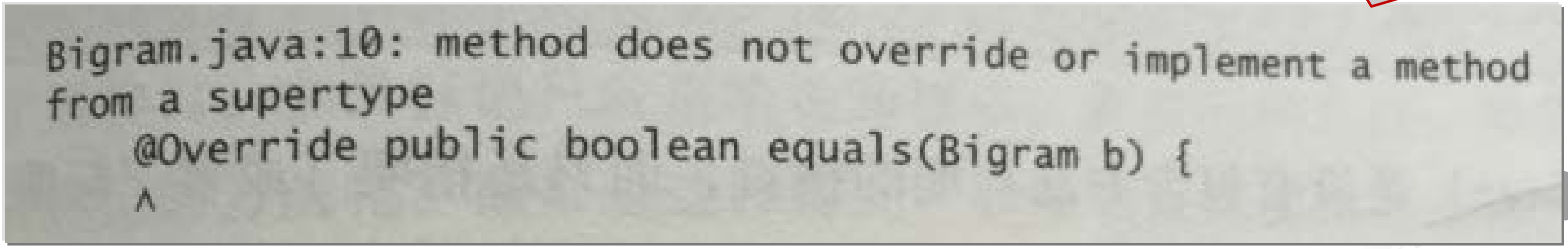
Override?
Overload?

NOTE

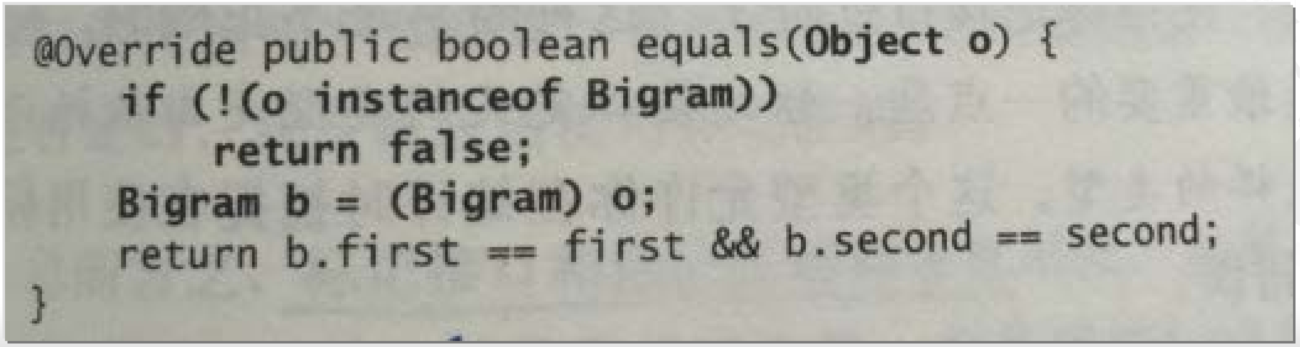
- 坚持使用Override注解
- Java1.5中增加Override注解，可防止前面的非法错误

```
@override public boolean equals(Bigram b){
        return b.first==first && b.second==second;
}
```

编译错误



```
Bigram.java:10: method does not override or implement a method
from a supertype
    @Override public boolean equals(Bigram b) {
    ^
```



```
@Override public boolean equals(Object o) {
    if (!(o instanceof Bigram))
        return false;
    Bigram b = (Bigram) o;
    return b.first == first && b.second == second;
}
```

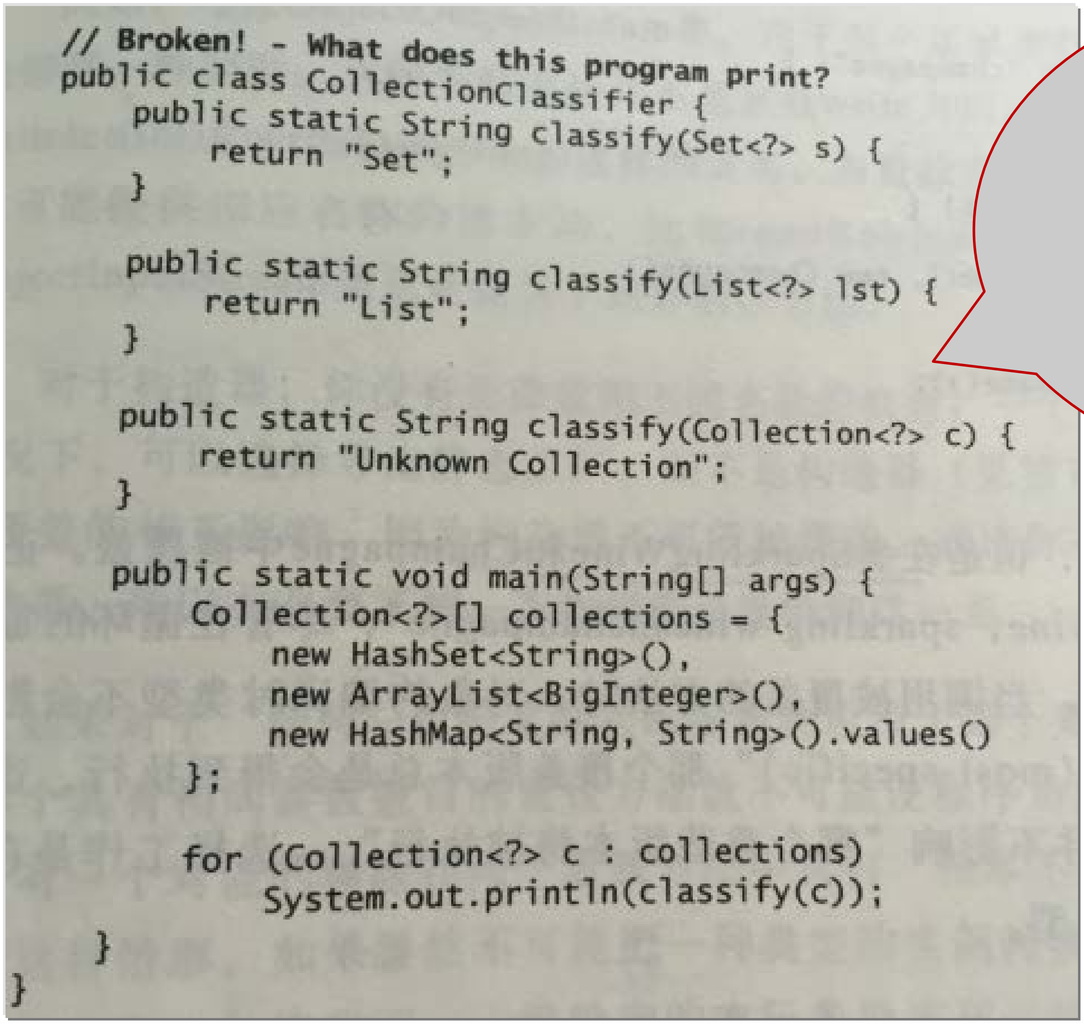
Overriding vs. Overloading

```java
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);
  }
}
```

```java
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);
  }
}
```

方法重写发生在通过继承而相关的不同类中；而方法重装可以发生在同一个类中，也可以发生在由于继承而相关的不同类中。

- overload(重载)方法的选择是静态的，而对于被覆盖的方法(overridden method)的选择则是动态的。



```
// Broken! - What does this program print?
public class CollectionClassifier {
    public static String classify(Set<?> s) {
        return "Set";
    }

    public static String classify(List<?> lst) {
        return "List";
    }

    public static String classify(Collection<?> c) {
        return "Unknown Collection";
    }

    public static void main(String[] args) {
        Collection<?>[] collections = {
            new HashSet<String>(),
            new ArrayList<BigInteger>(),
            new HashMap<String, String>().values()
        };

        for (Collection<?> c : collections)
            System.out.println(classify(c));
    }
}
```

打印"Unknown Collection" 三次

对于参数的匹配，overload，静态匹配

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*. Therefore, you can say that **Circle** is a subtype of **GeometricObject** and **GeometricObject** is a supertype for **Circle**.

PolymorphismDemo

Run

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*.

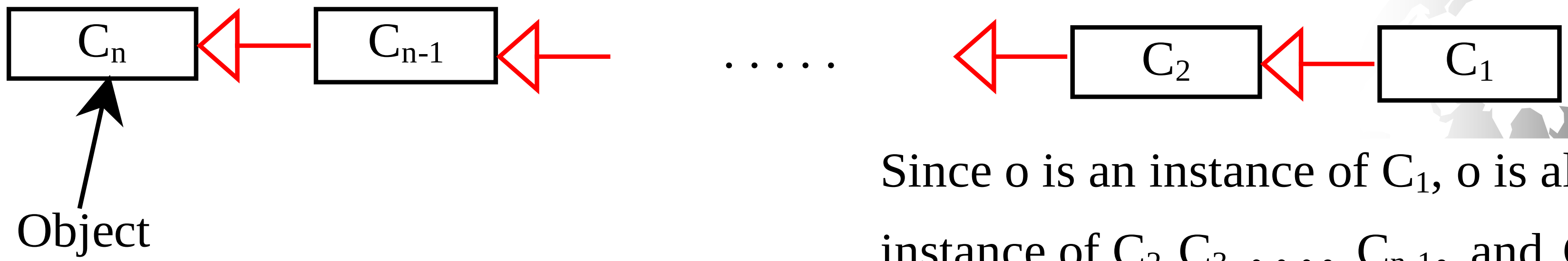
Animation

DynamicBindingDemo

Run

Dynamic Binding

Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, ..., 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, ..., 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, . . . , 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

You 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**. 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);
```

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());
  ...
}
```

TIP

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.

Java VS. C++

- Manager boss = (Manager)staff; //java
- 类似于C++中的 Manager* boss = dynamic_cast<Manager*>(staff);
- 之前一个重要的区别在于：当类型转换失败时，Java不会生成一个null对象，而是抛出一个异常；而在C++中，则会生成null对象，这样可以在一个操作中完成类型测试和类型转换

```
Manager *boss = dynamic_cast<Manager*>staff;
If(boss !=NULL)….
```

在Java中，则需要用instanceof和类型转换结合起来使用：

```
If(staff instanceof Manager)
{
    Manager boss = (Manager)staff;
}
```

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 Run

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;
}
```

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 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.

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.

Generic Type

ArrayList is known as a generic class with a generic type E. You can specify a concrete type to replace E when creating an ArrayList. For example, the following statement creates an ArrayList and assigns its reference to variable cities. This ArrayList object can be used to store strings.

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

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

TestArrayList Run

Differences and Similarities between Arrays and ArrayList

Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new ArrayList<>();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

DistinctNumbers

Run

Array Lists from/to Arrays

Creating an ArrayList from an array of objects:

```
 String[] array = {"red", "green", "blue"};
   ArrayList<String> list = new
ArrayList<>(Arrays.asList(array));
```

Creating an array of objects from an ArrayList:

```
   String[] array1 = new String[list.size()];
   list.toArray(array1);
```

max and min in an Array List

```
String[] array = {"red", "green", "blue"};
System.out.pritnln(java.util.Collections.max(
   new ArrayList<String>(Arrays.asList(array)));


String[] array = {"red", "green", "blue"};
System.out.pritnln(java.util.Collections.min(
  new ArrayList<String>(Arrays.asList(array)));
```

Shuffling an Array List

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new
   ArrayList<>(Arrays.asList(array));
java.util.Collections.shuffle(list);
System.out.println(list);
```

Sorting an Array List

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new
   ArrayList<>(Arrays.asList(array));
java.util.Collections.sort(list);
System.out.println(list);
```

The MyStack Classes

A stack to hold objects.

MyStack

MyStack	
-list: ArrayList	A list to store elements.
+isEmpty(): boolean	Returns true if this stack is empty.
+getSize(): int	Returns the number of elements in this stack.
+peek(): Object	Returns the top element in this stack.
+pop(): Object	Returns and removes the top element in this stack.
+push(o: Object): void	Adds a new element to the top of this stack.
+search(o: Object): int	Returns the position of the first element in the stack from the top that matches the specified element.

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.

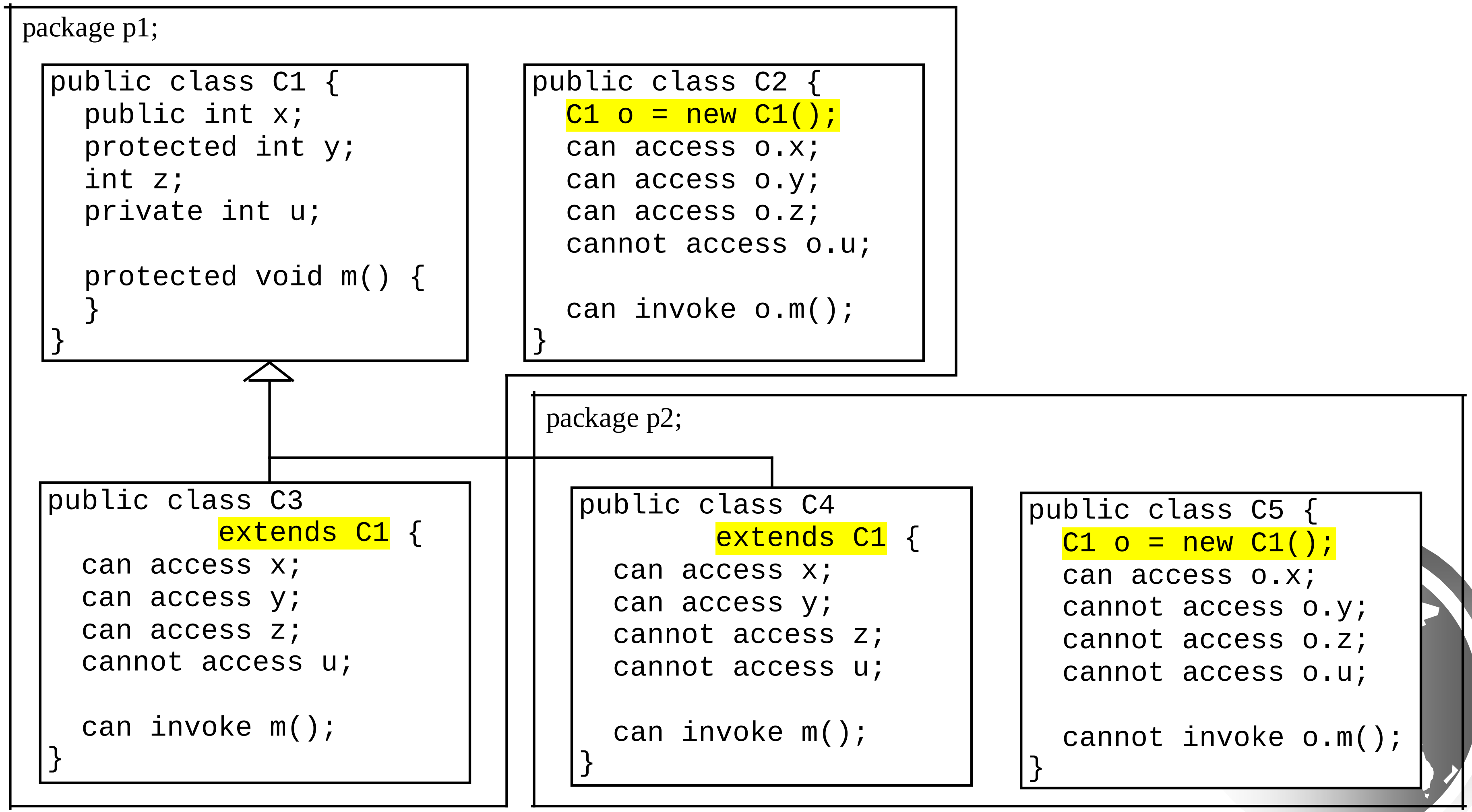
- private, default, protected, public

```
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



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();
}
```

```
public class C3
          extends C1 {
  can access x;
  can access y;
  can access z;
  cannot access u;

  can invoke m();
}
```

package p2;

```
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();
}
```

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.

复合优先于继承

- 继承是实现代码重用的有力手段，但它并非是完成这项工作的最佳手段。
- 原因在于：继承打破了封装性，也就是子类依赖于其超类中特定功能的实现。而超类的实现有可能会随着发行版本的不同而有所变化。如果真的发生了变化，则子类有可能遭到破坏。
- 例子：假设有一个程序使用HashSet。为了调优该程序的性能，需要查询HashSet一共添加过多少元素。

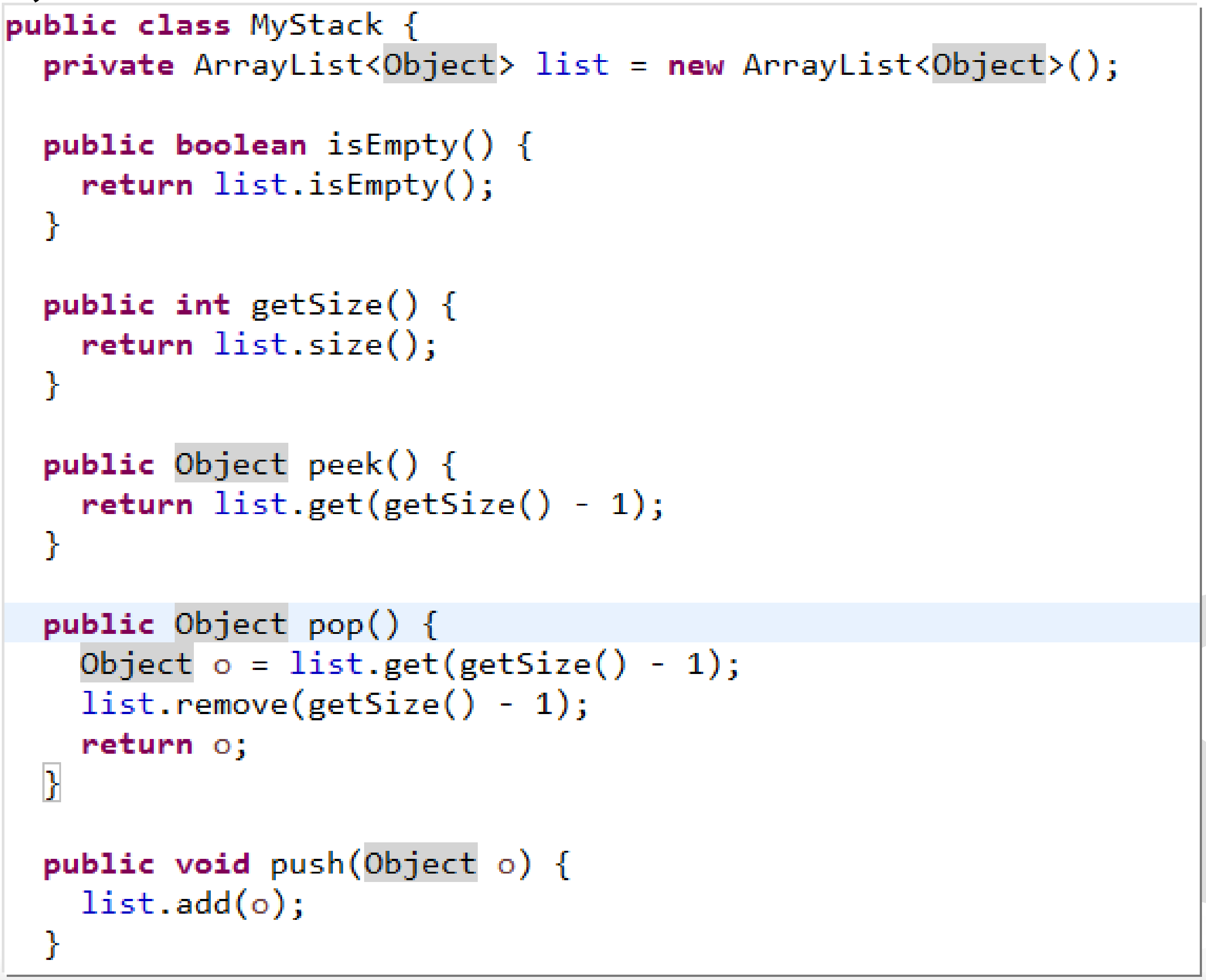
```
public class InstrumentedHashSet<E> extends HashSet<E>{
    private int addCount = 0;
    public InstrumentedHashSet(){}
    public InstrumentedHashSet(int initCap, float loadFactor){
            super(initCap, loadFactor);
    }
    @override public boolean add(E e){
            addCount++;
            return super.add(e);
    }
    @override public boolean addAll(Collection<? Extends E> c){
            addCount += c.size();
            return super.addAll(c);
    }
    public int getAddCount(){
            return addCount;
    }
}
```

```
InstrumentedHashSet<String> s = new InstrumentedHashSet<String>();
s.addAll(Arrays.asList(“snap”, “Crackle”, “Pop”));
```

getAddCount()返回6

因为addAll内部是add来实现的。

- 复合（ composition）：不继承现有的类，而是在新的类中增加一个私有域，并引用现有类的一个实例。



```java
public class MyStack {
  private ArrayList<Object> list = new ArrayList<Object>();

  public boolean isEmpty() {
    return list.isEmpty();
  }

  public int getSize() {
    return list.size();
  }

  public Object peek() {
    return list.get(getSize() - 1);
  }

  public Object pop() {
    Object o = list.get(getSize() - 1);
    list.remove(getSize() - 1);
    return o;
  }

  public void push(Object o) {
    list.add(o);
  }
```

嵌套类（nested class)

- 可以在一个类的内部定义另一个类，这种类称为嵌套类（nested classes），它有两种类型：静态嵌套类和非静态嵌套类。静态嵌套类使用很少，最重要的是非静态嵌套类，也即是被称作为内部类（inner）。嵌套类从JDK1.1开始引入。其中inner类又可分为三种：
  - 其一、在一个类（外部类）中直接定义的内部类；
  - 其二、在一个方法（外部类的方法）中定义的局部类(local class)；
  - 其三、匿名内部类(anonymous class)。

Simple Uses of Inner Classes

- **Inner classes** are classes defined within other classes
  - The class that includes the inner class is called the **outer class**
  - There is no particular location where the definition of the inner class (or classes) must be place within the outer class
  - Placing it first or last, however, will guarantee that it is easy to find

Simple Uses of Inner Classes

- An inner class definition is a member of the outer class in the same way that the instance variables and methods of the outer class are members
  - An inner class is local to the outer class definition
  - The name of an inner class may be reused for something else outside the outer class definition
  - If the inner class is private, then the inner class cannot be accessed by name outside the definition of the outer class

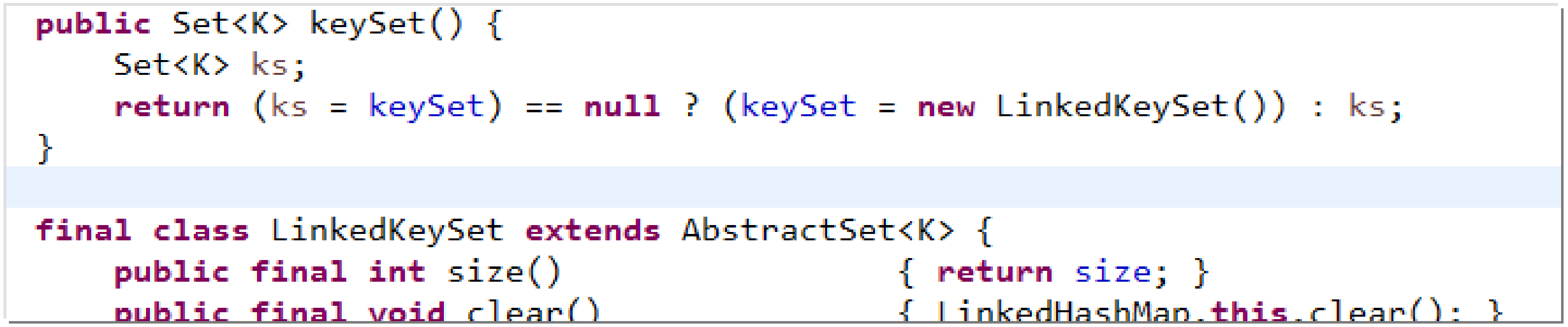
Inner/Outer Classes

```
public class Outer
{
      private class Inner
      {
            // inner class instance variables
            // inner class methods

      } // end of inner class definition


      // outer class instance variables
      // outer class methods
}
```

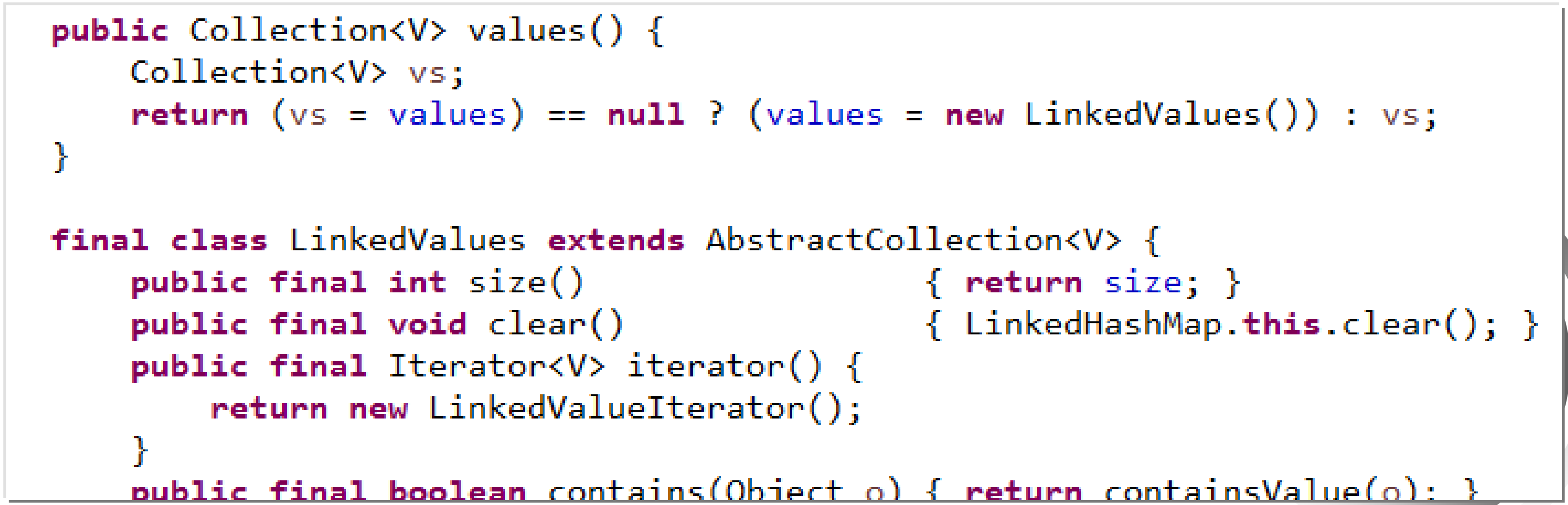
Simple Uses of Inner Classes

- There are two main advantages to inner classes
  - They can make the outer class more self-contained since they are defined inside a class
  - Both of their methods have access to **each other's** private methods and instance variables
- Using an inner class as a helping class is one of the most useful applications of inner classes
  - If used as a helping class, an inner class should be marked private

LinkedHashMap类中有LinkedKeySet和LinkedValues内部类，可以帮助操作KeySet和Values



```
public Set<K> keySet() {
    Set<K> ks;
    return (ks = keySet) == null ? (keySet = new LinkedKeySet()) : ks;
}

final class LinkedKeySet extends AbstractSet<K> {
    public final int size()                 { return size; }
    public final void clear()               { LinkedHashMap.this.clear(); }
```

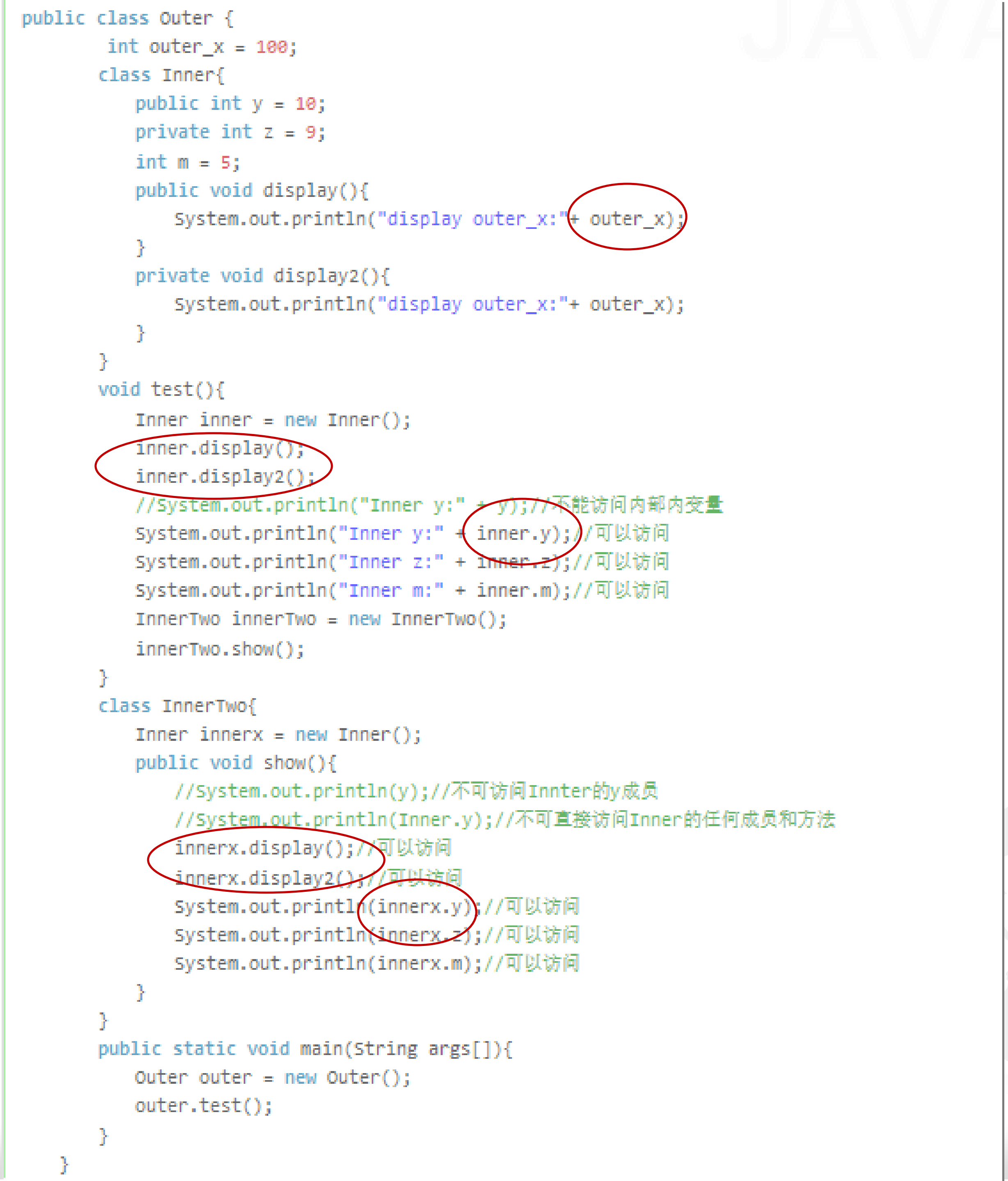


```
public Collection<V> values() {
    Collection<V> vs;
    return (vs = values) == null ? (values = new LinkedValues()) : vs;
}

final class LinkedValues extends AbstractCollection<V> {
    public final int size()                 { return size; }
    public final void clear()               { LinkedHashMap.this.clear(); }
    public final Iterator<V> iterator() {
        return new LinkedValueIterator();
    }
    public final boolean contains(Object o) { return containsValue(o); }
```

Inner and Outer Classes Have Access to Each Other's Private Members

- Within the definition of a method of an inner class:
  - It is legal to reference a private instance variable of the outer class
  - It is legal to invoke a private method of the outer class
  - Essentially, the inner class has a hidden reference to the outer class
- Within the definition of a method of the outer class
  - It is legal to reference a private instance variable of the inner class on an object of the inner class
  - It is legal to invoke a (nonstatic) method of the inner class <u>as long as an object of the inner class is used as a calling object</u>

- Within the definition of the inner or outer classes, the modifiers **`public`** and **`private`** are equivalent



```
public class Outer {
        int outer_x = 100;
       class Inner{
           public int y = 10;
           private int z = 9;
           int m = 5;
           public void display(){
               System.out.println("display outer_x:"+ outer_x);
           }
           private void display2(){
               System.out.println("display outer_x:"+ outer_x);
           }
       }
       void test(){
           Inner inner = new Inner();
           inner.display();
           inner.display2();
           //System.out.println("Inner y:" + y);//不能访问内部内变量
           System.out.println("Inner y:" + inner.y);//可以访问
           System.out.println("Inner z:" + inner.z);//可以访问
           System.out.println("Inner m:" + inner.m);//可以访问
           InnerTwo innerTwo = new InnerTwo();
           innerTwo.show();
       }
       class InnerTwo{
           Inner innerx = new Inner();
           public void show(){
               //System.out.println(y);//不可访问Innter的y成员
               //System.out.println(Inner.y);//不可直接访问Inner的任何成员和方法
               innerx.display();//可以访问
               innerx.display2();//可以访问
               System.out.println(innerx.y);//可以访问
               System.out.println(innerx.z);//可以访问
               System.out.println(innerx.m);//可以访问
           }
       }
       public static void main(String args[]){
           Outer outer = new Outer();
           outer.test();
       }
   }
```

Class with an Inner Class

Display 13.9 Class with an Inner Class *(Part 1 of 2)*

```
public class BankAccount
{
    private class Money
    {
        private long dollars;
        private int cents;

        public Money(String stringAmount)
        {
            abortOnNull(stringAmount);
            int length = stringAmount.length();
            dollars = Long.parseLong(
                         stringAmount.substring(0, length - 3));
            cents = Integer.parseInt(
                         stringAmount.substring(length - 2, length));
        }

        public String getAmount()
        {
            if (cents > 9)
               return (dollars + "." + cents);
            else
               return (dollars + ".0" + cents);
        }
```

The modifier `private` in this line should not be changed to `public`.

However, the modifiers `public` and `private` inside the inner class `Money` can be changed to anything else and it would have no effect on the class `BankAccount`.

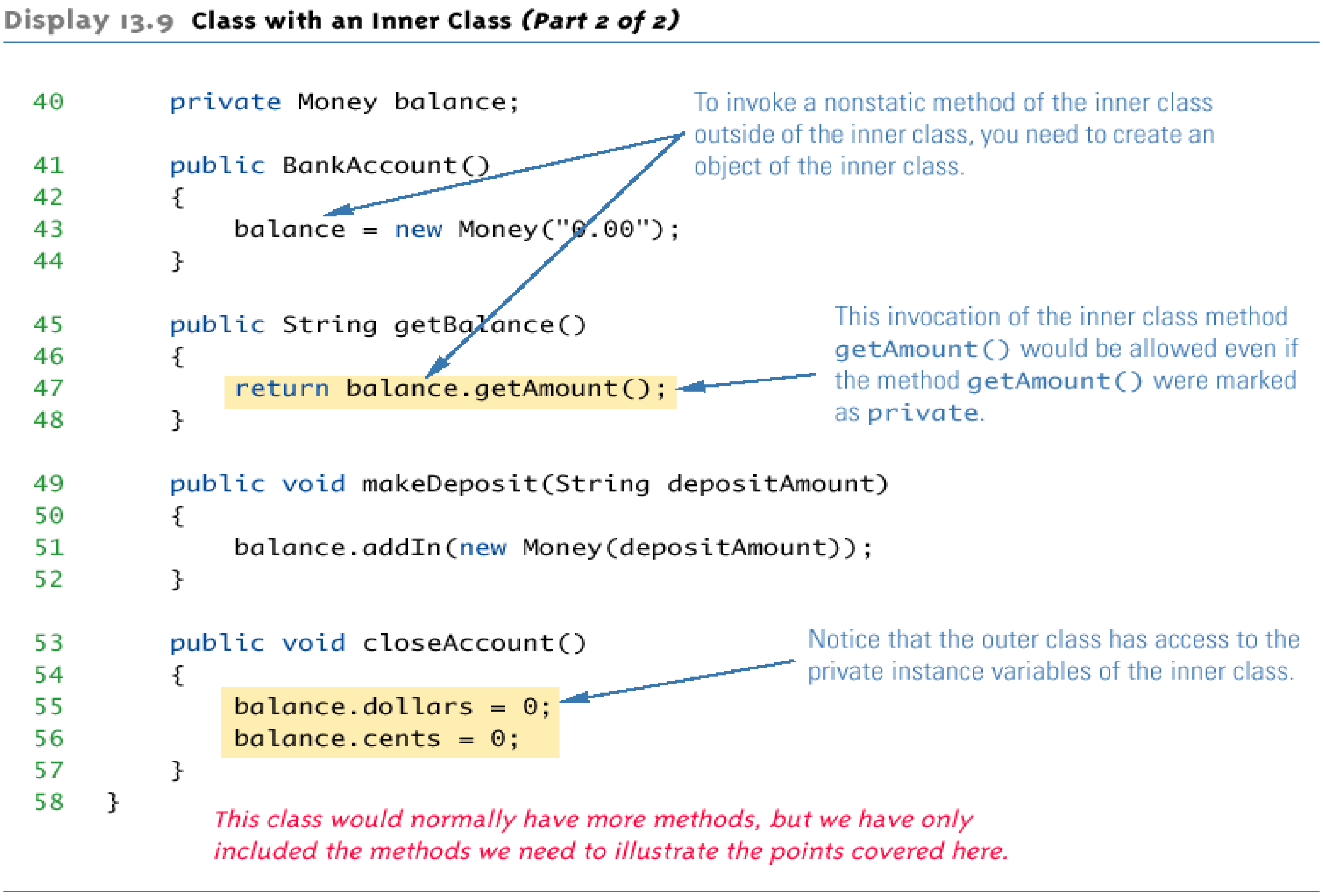
Class with an Inner Class

Display 13.9 Class with an Inner Class *(Part 1 of 2)* (continued)

```
    public void addIn(Money secondAmount)
    {
        abortOnNull(secondAmount);
        int newCents = (cents + secondAmount.cents)%100;
        long carry = (cents + secondAmount.cents)/100;
        cents = newCents;
        dollars = dollars + secondAmount.dollars + carry;
    }

   private void abortOnNull(Object o)
   {
       if (o == null)
       {
           System.out.println("Unexpected null argument.");
           System.exit(0);
       }
   }
}
```

The definition of the inner class ends here, but the definition of the outer class continues in Part 2 of this display.

Class with an Inner Class



Display 13.9 Class with an Inner Class *(Part 2 of 2)*

```
    private Money balance;

    public BankAccount()
    {
        balance = new Money("0.00");
    }

    public String getBalance()
    {
        return balance.getAmount();
    }

    public void makeDeposit(String depositAmount)
    {
        balance.addIn(new Money(depositAmount));
    }

    public void closeAccount()
    {
        balance.dollars = 0;
        balance.cents = 0;
    }
}
```

To invoke a nonstatic method of the inner class outside of the inner class, you need to create an object of the inner class.

This invocation of the inner class method `getAmount()` would be allowed even if the method `getAmount()` were marked as `private`.

Notice that the outer class has access to the private instance variables of the inner class.

This class would normally have more methods, but we have only included the methods we need to illustrate the points covered here.

Referring to a Method of the Outer Class

- If a method is invoked in an inner class （当内部类调用一个函数时。。。）
  - If the inner class has no such method, then it is assumed to be an invocation of the method of that name in the outer class
  - If both the inner and outer class have a method with the same name, then it is assumed to be an invocation of the method in the inner class
  - If both the inner and outer class have a method with the same name, and the intent is to invoke the method in the outer class, then the following invocation must be used:

```
OuterClassName.this.methodName()
```

Public Inner Classes

- If an inner class is marked **`public`**, then it can be used outside of the outer class
- In the case of a nonstatic inner class, it must be created using an object of the outer class

```
BankAccount account = new
 BankAccount();
BankAccount.Money amount =
   account.new Money("41.99");
```

  - Note that the prefix ***`account.`*** must come before ***`new`***
  - The new object **`amount`** can now invoke methods from the inner class, but only from the inner class

Public Inner Classes

- In the case of a static inner class, the procedure is similar to, but simpler than, that for nonstatic inner classes

```
OuterClass.InnerClass innerObject =
                 new
  OuterClass.InnerClass();
```

  – Note that all of the following are acceptable

```
innerObject.nonstaticMethod();
innerObject.staticMethod();
OuterClass.InnerClass.staticMethod();
```

Public Money Inner Class

If the Money inner class in the BankAccount example was defined as public, we can create and use objects of type Money outside the BankAccount class.

```
// this is okay in main( )
BankAccount account = new BankAccount( );
BankAccount.Money amt =    // note syntax
               account.new Money( “41.99” );
System.out.println( amt.getAmount( ) );
// but NOT this – why not??
System.out.println( amt.getBalance( ) );
```

Static Inner Classes

- A normal inner class has a connection between its objects and the outer class object that created the inner class object
  - This allows an inner class definition to reference an instance variable, or invoke a method of the outer class
- There are certain situations, however, when an inner class must be static
  - If an object of the inner class is created within a static method of the outer class
  - If the inner class must have static members

Static Inner Classes

- Since a static inner class has no connection to an object of the outer class, within an inner class method
  - Instance variables of the outer class **cannot be** referenced
  - Nonstatic methods of the outer class **cannot be** invoked
- To invoke a static method or to name a static variable of a static inner class within the outer class, preface each with the name of the inner class and a dot

Multiple Inner Classes

- A class can have as many inner classes as it needs.
- Inner classes have access to each other's private members as long as an object of the other inner class is used as the calling object.

The `.class` File for an Inner Class

- Compiling any class in Java produces a `.class` file named ***`ClassName.class`***
- Compiling a class with one (or more) inner classes causes both (or more) classes to be compiled, and produces two (or more) .class files
  - Such as ***`ClassName.class`*** **and** ***`ClassName$InnerClassName.class`***

Nesting Inner Classes

- It is legal to nest inner classes within inner classes
  - The rules are the same as before, but the names get longer
  - Given class **`A`**, which has public inner class **`B`**, which has public inner class **`C`**, then the following is valid:

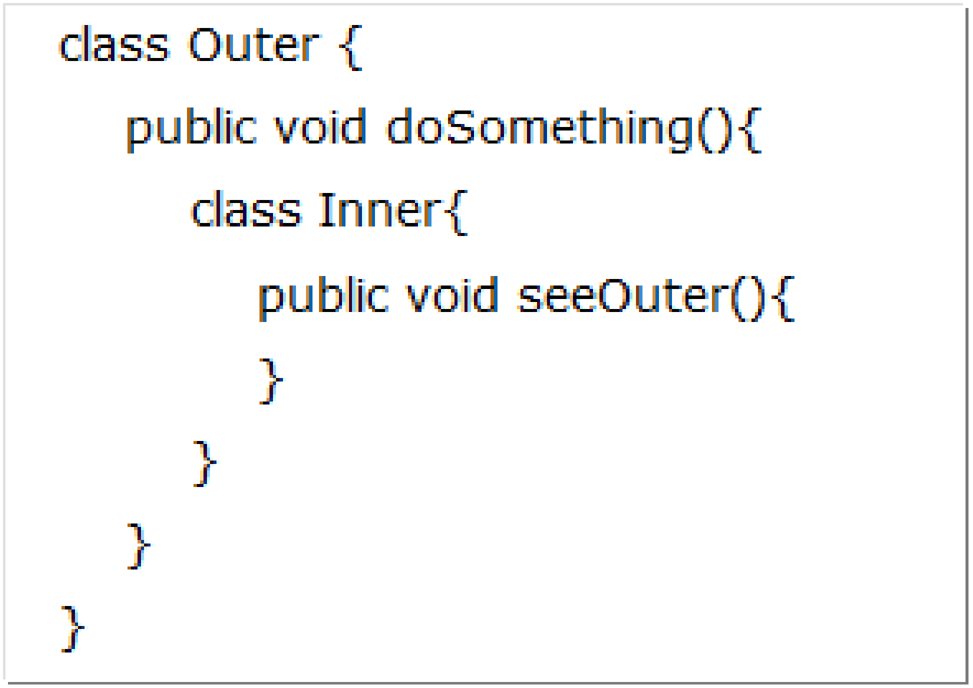
```
A aObject = new A();
A.B bObject = aObject.new B();
A.B.C cObject = bObject.new C();
```

Inner Classes and Inheritance

- Given an `OuterClass` that has an `InnerClass`
  - Any `DerivedClass` of `OuterClass` will automatically have `InnerClass` as an inner class
  - In this case, the `DerivedClass` cannot override the `InnerClass`
- An outer class can be a derived class
- An inner class can be a derived class also

Local class

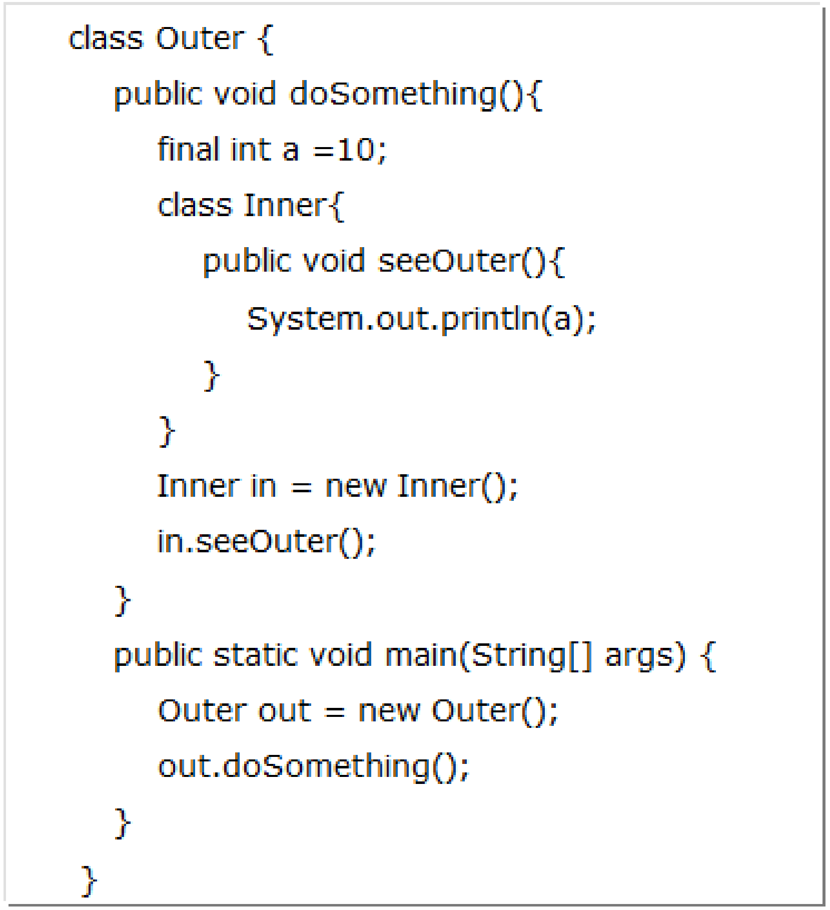
- 类定义在方法内：



```
class Outer {
    public void doSomething(){
        class Inner{
            public void seeOuter(){
            }
        }
    }
}
```

A、方法内部类只能在定义该内部类的方法内实例化，不可以在此方法外对其实例化。
B、方法内部类对象不能使用该内部类所在方法的非final局部变量。

- 方法的局部变量位于栈上，只存在于该方法的生命期内。当一个方法结束，其栈结构被删除，局部变量成为历史。但是该方法结束之后，在方法内创建的内部类对象可能仍然存在于堆中！例如，如果对它的引用被传递到其他某些代码，并存储在一个成员变量内。正因为不能保证局部变量的存活期和方法内部类对象的一样长， 所以内部类对象不能使用它们（非final的变量）。



```
class Outer {
    public void doSomething(){
        final int a =10;
        class Inner{
            public void seeOuter(){
                System.out.println(a);
            }
        }
        Inner in = new Inner();
        in.seeOuter();
    }
    public static void main(String[] args) {
        Outer out = new Outer();
        out.doSomething();
    }
}
```

- 方法内部类的修饰符
  与成员内部类不同，方法内部类更像一个局部变量。
  可以用于修饰方法内部类的只有final和abstract。
- 静态方法内的方法内部类
  静态方法是没有this引用的，因此在静态方法内的内部类遭受同样的待遇，即：只能访问外部类的静态成员。

Anonymous Classes

- If an object is to be created, but there is no need to name the object's class, then an *anonymous class* definition can be used
  - The class definition is embedded inside the expression with the **`new`** operator
  - An anonymous class is an abbreviated notation for creating a simple local object "in-line" within any expression, simply by wrapping the desired code in a "new" expression.
- Anonymous classes are sometimes used when they are to be assigned to a variable of another type
  - The other type must be such that an object of the anonymous class is also an object of the other type
  - The other type is usually a Java interface
  - Not every inner class should be anonymous, but very simple "one-shot" local objects are such a common case that they merit some syntactic sugar.

Anonymous Classes

Display 13.11 Anonymous Classes (*Part 1 of 2*)

This is just a toy example to demonstrate the Java syntax for anonymous classes.

```java
public class AnonymousClassDemo
{
    public static void main(String[] args)
    {
        NumberCarrier anObject =
                       new NumberCarrier()
                       {
                           private int number;
                           public void setNumber(int value)
                           {
                               number = value;
                           }
                           public int getNumber()
                           {
                              return number;
                           }
                        };
```

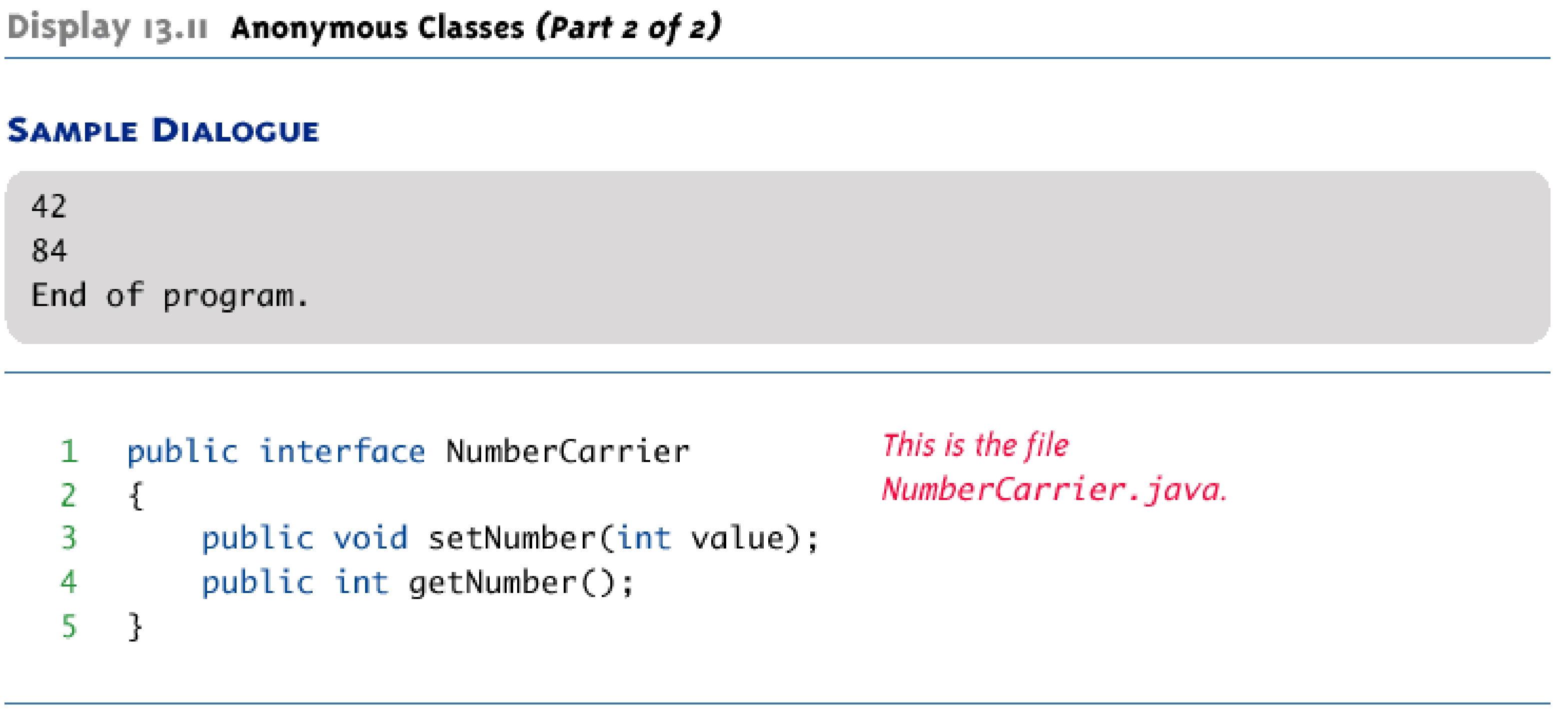
Anonymous Classes

Display 13.11 Anonymous Classes *(Part 1 of 2)*

```
        NumberCarrier anotherObject =
                  new NumberCarrier()
                  {
                      private int number;
                      public void setNumber(int value)
                      {
                          number = 2*value;
                      }
                      public int getNumber()
                      {
                          return number;
                      }
                  };
        anObject.setNumber(42);
        anotherObject.setNumber(42);
        showNumber(anObject);
        showNumber(anotherObject);
        System.out.println("End of program.");
    }
    public static void showNumber(NumberCarrier o)
    {
        System.out.println(o.getNumber());
    }
}
```

This is still the file AnonymousClassDemo.java.

Anonymous Classes



Display 13.11 **Anonymous Classes *(Part 2 of 2)***

SAMPLE DIALOGUE

```
42
84
End of program.
```

```java
public interface NumberCarrier
{
    public void setNumber(int value);
    public int getNumber();
}
```

This is the file NumberCarrier.java.

Inner Class Listeners

A listener class is designed specifically to create a listener object for a GUI component (e.g., a button). It will not be shared by other applications. So, it is appropriate to define the listener class inside the frame class as an inner class.

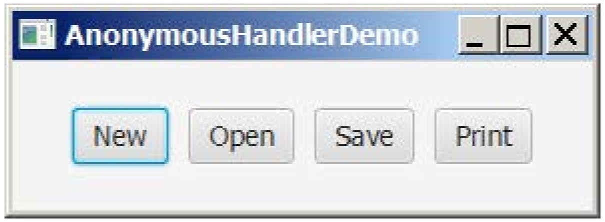
Anonymous Inner Classes

```
public void start(Stage primaryStage) {
  // Omitted

  btEnlarge.setOnAction(
    new EnlargeHandler());
}

class EnlargeHandler
    implements EventHandler<ActionEvent> {
  public void handle(ActionEvent e) {
    circlePane.enlarge();
  }
}
```

(a) Inner class EnlargeListener

```
public void start(Stage primaryStage) {
  // Omitted

  btEnlarge.setOnAction(
    new ~~class EnlargeHandlner~~
      ~~implements~~ EventHandler<ActionEvent>() {
      public void handle(ActionEvent e) {
        circlePane.enlarge();
      }
    });
}
```

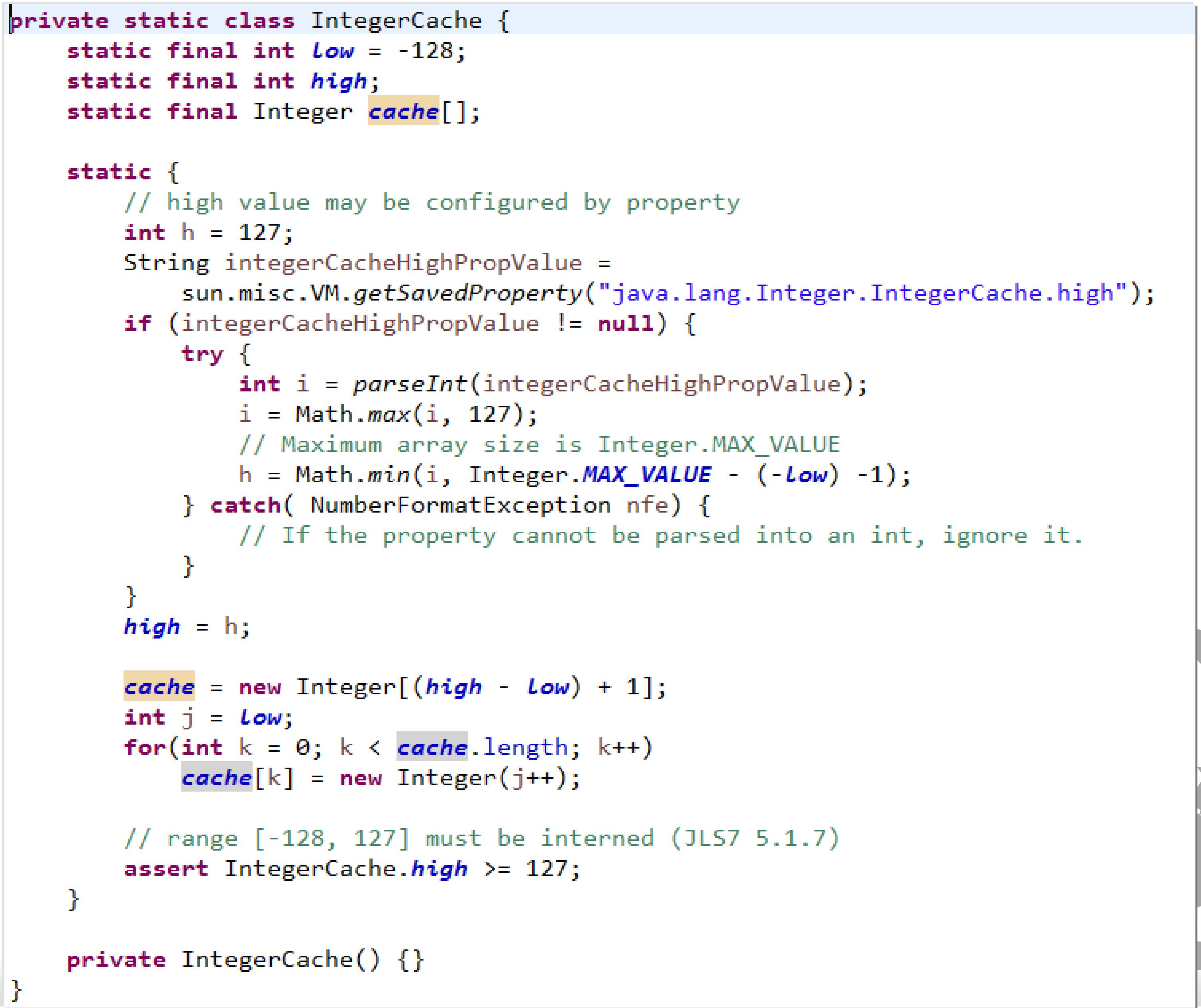
(b) Anonymous inner class



AnonymousHandlerDemo

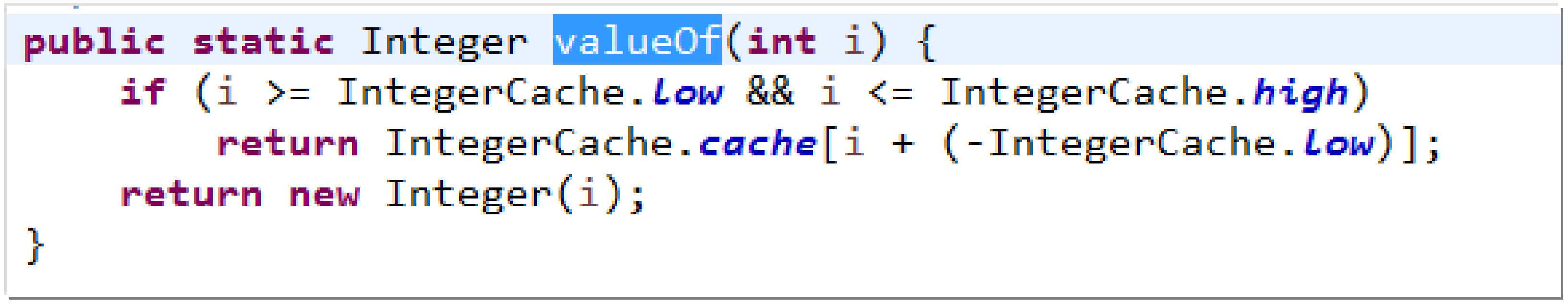
Run

实际例子，Integer的静态嵌套类：



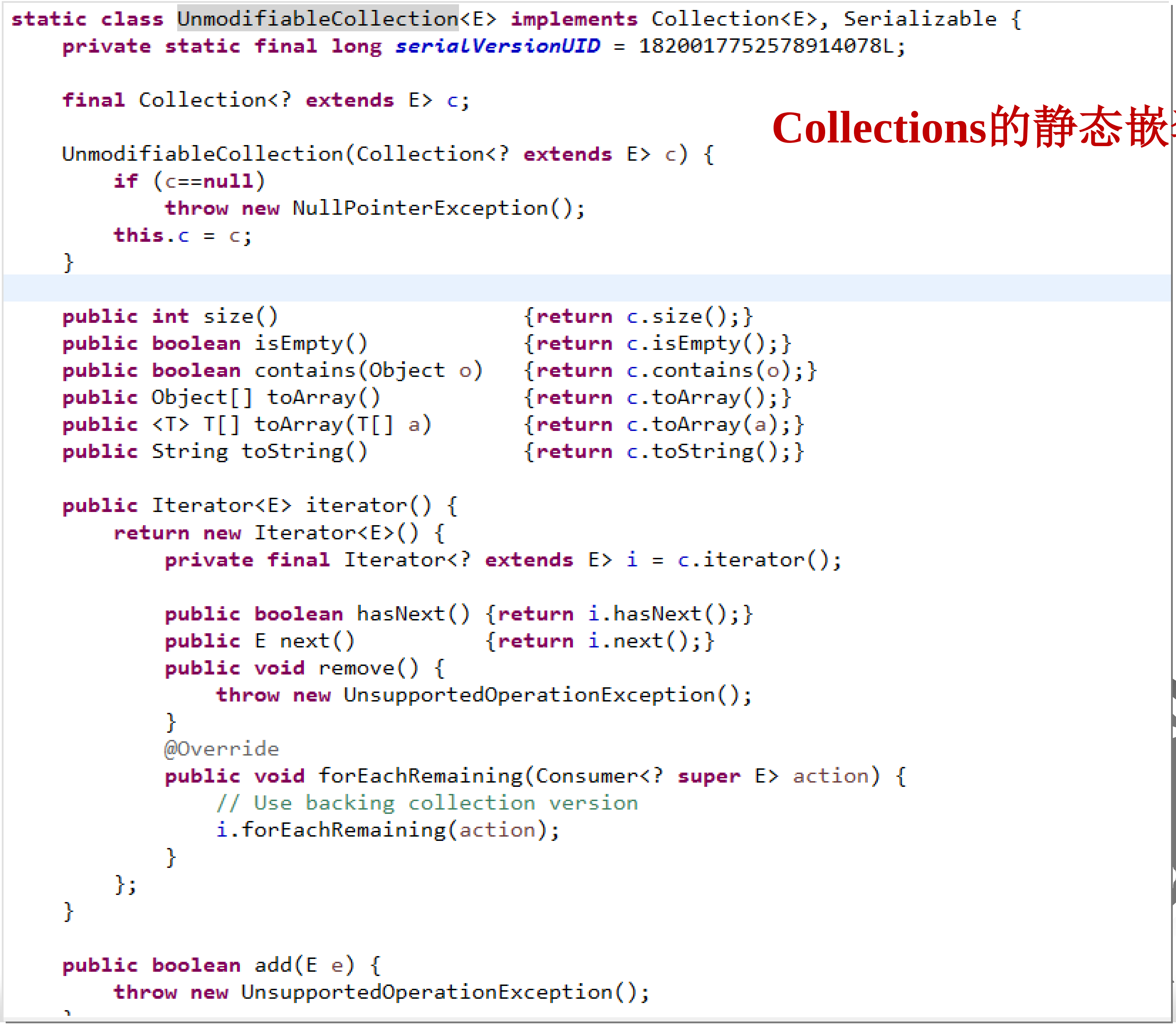
```java
private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static final Integer cache[];

    static {
        // high value may be configured by property
        int h = 127;
        String integerCacheHighPropValue =
            sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
        if (integerCacheHighPropValue != null) {
            try {
                int i = parseInt(integerCacheHighPropValue);
                i = Math.max(i, 127);
                // Maximum array size is Integer.MAX_VALUE
                h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
            } catch( NumberFormatException nfe) {
                // If the property cannot be parsed into an int, ignore it.
            }
        }
        high = h;

        cache = new Integer[(high - low) + 1];
        int j = low;
        for(int k = 0; k < cache.length; k++)
            cache[k] = new Integer(j++);

        // range [-128, 127] must be interned (JLS7 5.1.7)
        assert IntegerCache.high >= 127;
    }

    private IntegerCache() {}
}
```

- Integer类中



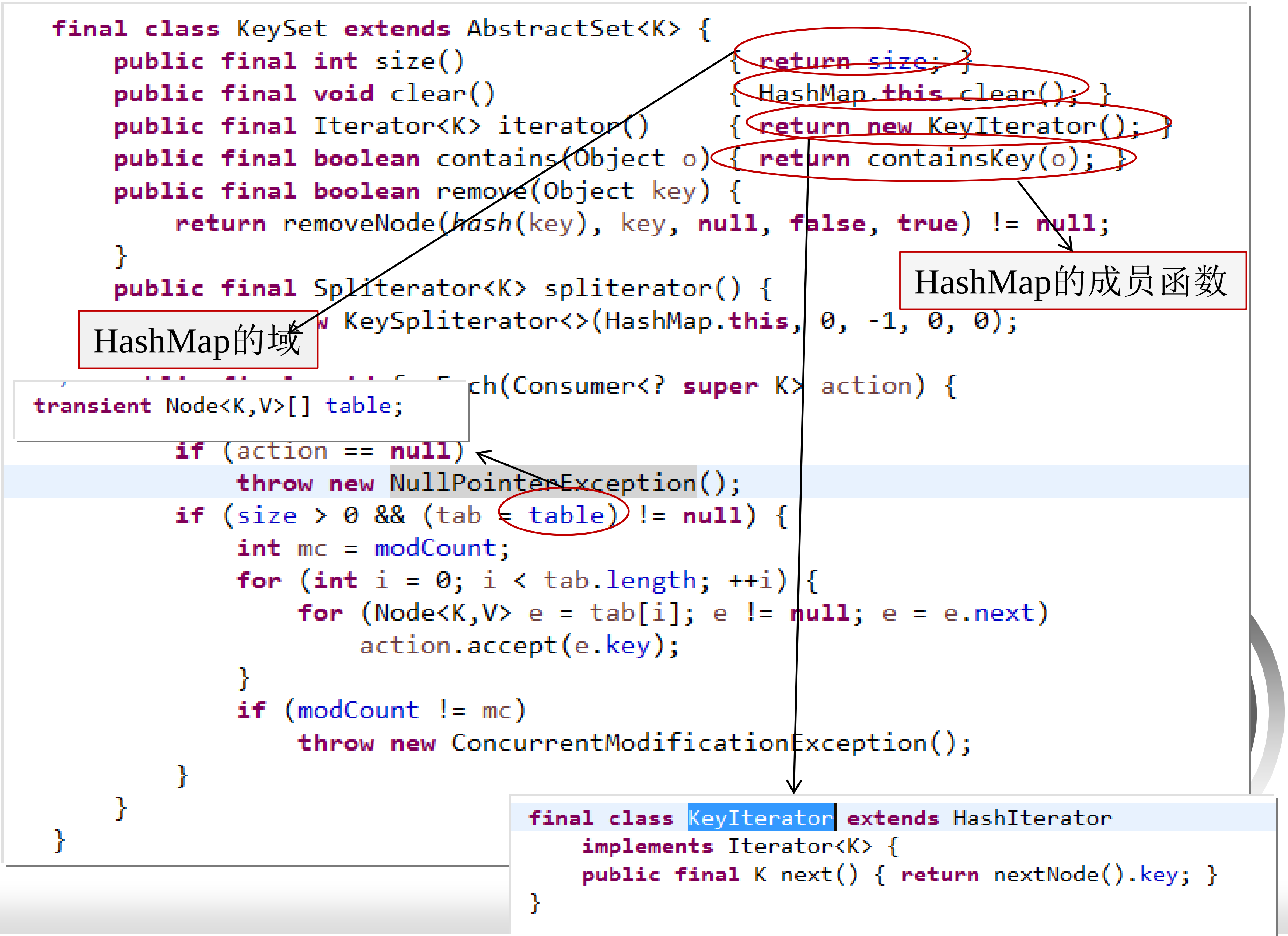
```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

Collections的静态嵌套类



```
static class UnmodifiableCollection<E> implements Collection<E>, Serializable {
    private static final long serialVersionUID = 1820017752578914078L;

    final Collection<? extends E> c;

    UnmodifiableCollection(Collection<? extends E> c) {
        if (c==null)
            throw new NullPointerException();
        this.c = c;
    }

    public int size()                   {return c.size();}
    public boolean isEmpty()            {return c.isEmpty();}
    public boolean contains(Object o)   {return c.contains(o);}
    public Object[] toArray()           {return c.toArray();}
    public <T> T[] toArray(T[] a)       {return c.toArray(a);}
    public String toString()            {return c.toString();}

    public Iterator<E> iterator() {
        return new Iterator<E>() {
            private final Iterator<? extends E> i = c.iterator();

            public boolean hasNext() {return i.hasNext();}
            public E next()          {return i.next();}
            public void remove() {
                throw new UnsupportedOperationException();
            }
            @Override
            public void forEachRemaining(Consumer<? super E> action) {
                // Use backing collection version
                i.forEachRemaining(action);
            }
        };
    }

    public boolean add(E e) {
        throw new UnsupportedOperationException();
```

HashMap中的成员内部类



```
final class KeySet extends AbstractSet<K> {
    public final int size()                 { return size; }
    public final void clear()               { HashMap.this.clear(); }
    public final Iterator<K> iterator()     { return new KeyIterator(); }
    public final boolean contains(Object o) { return containsKey(o); }
    public final boolean remove(Object key) {
        return removeNode(hash(key), key, null, false, true) != null;
    }
    public final Spliterator<K> spliterator() {
                   w KeySpliterator<>(HashMap.this, 0, -1, 0, 0);

                          ch(Consumer<? super K> action) {
        if (action == null)
            throw new NullPointerException();
        if (size > 0 && (tab = table) != null) {
            int mc = modCount;
            for (int i = 0; i < tab.length; ++i) {
                for (Node<K,V> e = tab[i]; e != null; e = e.next)
                    action.accept(e.key);
            }
            if (modCount != mc)
                throw new ConcurrentModificationException();
        }
    }
}
```

```
transient Node<K,V>[] table;
```

```
final class KeyIterator extends HashIterator
    implements Iterator<K> {
    public final K next() { return nextNode().key; }
}
```

```java
final class Values extends AbstractCollection<V> {
    public final int size()                 { return size; }
    public final void clear()               { HashMap.this.clear(); }
    public final Iterator<V> iterator()     { return new ValueIterator(); }
    public final boolean contains(Object o) { return containsValue(o); }
    public final Spliterator<V> spliterator() {
        return new ValueSpliterator<>(HashMap.this, 0, -1, 0, 0);
    }
    public final void forEach(Consumer<? super V> action) {
        Node<K,V>[] tab;
        if (action == null)
            throw new NullPointerException();
        if (size > 0 && (tab = table) != null) {
            int mc = modCount;
            for (int i = 0; i < tab.length; ++i) {
                for (Node<K,V> e = tab[i]; e != null; e = e.next)
                    action.accept(e.value);
            }
            if (modCount != mc)
                throw new ConcurrentModificationException();
        }
    }
}
```