23. Polymorphism

Java

Summer 2008

Instructor: Dr. Masoud Yaghini

Outline

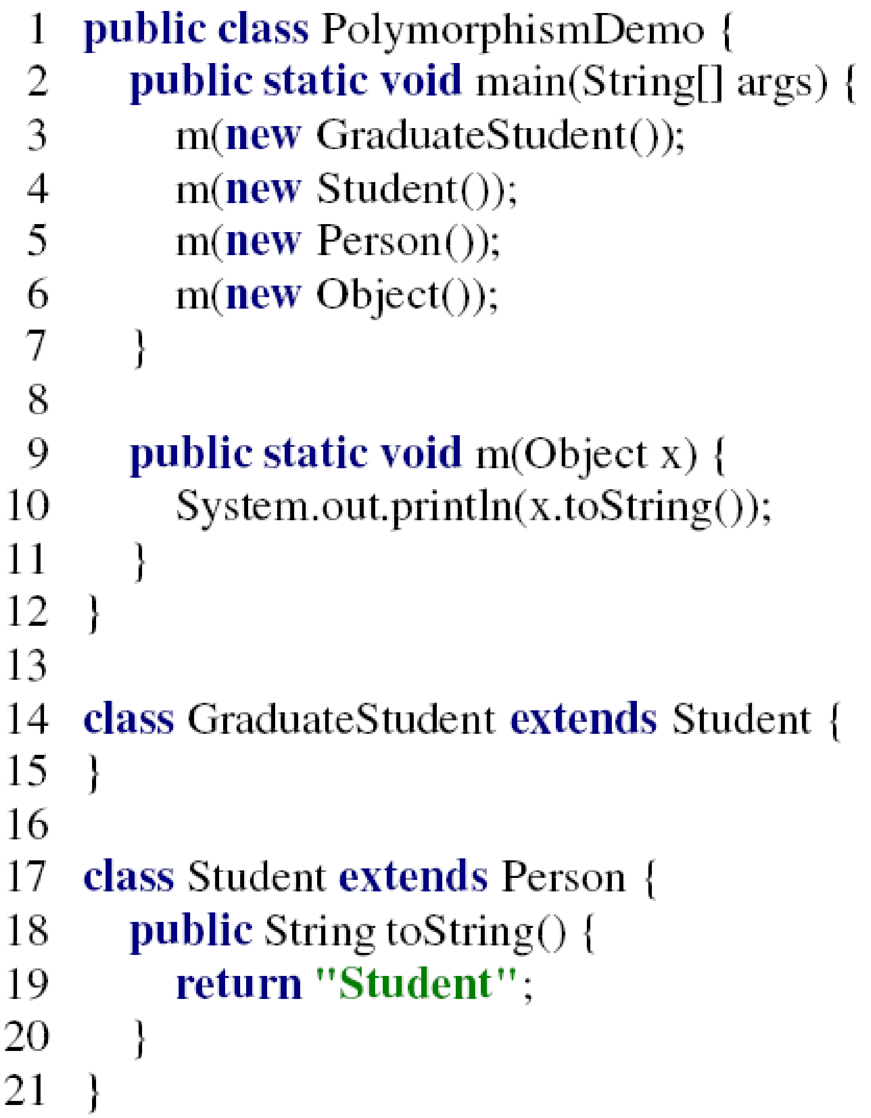
- Polymorphism, Dynamic Binding, and Generic Programming
- Casting Objects and the instanceof Operator
- The ArrayList Class
- The protected Data and Methods
- The final Classes, Methods, and Variables
- The this Keyword
- Getting Input from the Console
- References

Polymorphism, Dynamic Binding, and Generic Programming

Polymorphism

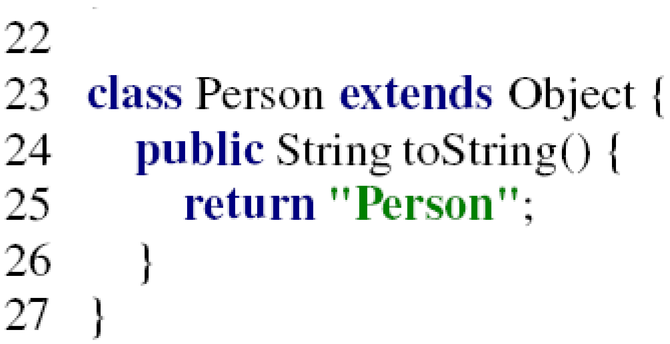
- The inheritance relationship enables a subclass to inherit features from its superclass with additional new features.
- A subclass is a specialization of its superclass
- Every instance of a subclass is an instance of its superclass, but not vice versa.
- For example, every circle is an object, but not every object is a circle.
- Therefore, you can always pass an instance of a subclass to a parameter of its superclass type.

PolymorphismDemo.java



```java
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";
    }
}
```

PolymorphismDemo.java



```java

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

- The output?

```
Student
Student
Person
java.lang.Object@10b30a7
```

Polymorphism

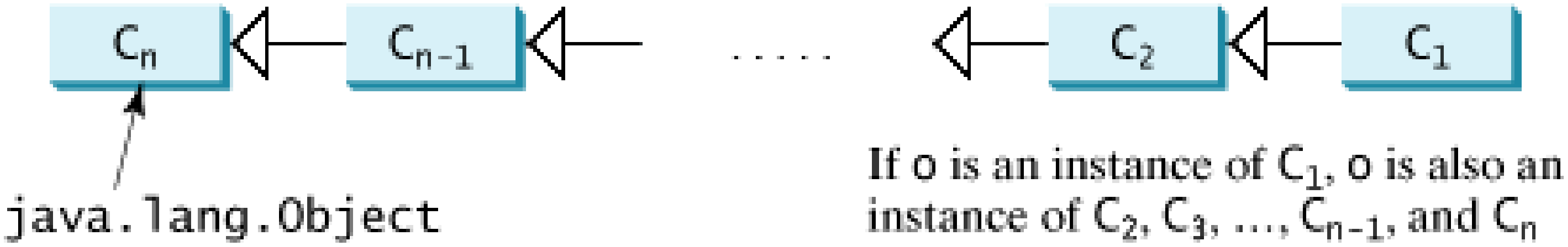
- When the method m(Object x) is executed, the argument x's toString method is invoked.
- x may be an instance of GraduateStudent, Student, Person, or Object.
- Classes GraduateStudent, Student, Person, and Object have their own implementations of the toString method.
- Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime.

Polymorphism

- This capability is known as ***dynamic binding***. It is also known as ***polymorphism*** (from a Greek word meaning "many forms") because one method has many implementations.
- Polymorphism is a feature that an object of a subtype can be used wherever its supertype value is required.

Polymorphism

- 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, as shown below:


C_n
C_{n-1}
.....
C_2
C_1
java.lang.Object
If o is an instance of C_1, o is also an instance of C_2, C_3, ..., C_{n-1}, and 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.

Polymorphism

- 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.
- For example, when m(new GraduateStudent()) is invoked, the toString method defined in the Student class is used.

Generic Programming

- 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 invoked (e.g., toString) is determined dynamically.

Casting Objects and the instanceof Operator

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);

Casting Objects

- The statement Object o = new Student(), known as ***implicit casting***, is legal because an instance of Student is automatically an instance of Object.
- Suppose you want to assign the object reference o to a variable of the type Student type using the following statement:

  Student b = o;
- A compilation error would occur. Why does the statement Object o = new Student() work and the statement Student b = o doesn't?
- Because a Student object is always an instance of Object, but an Object is not necessarily an instance of Student.

Casting Objects

- Even though you can see that o is really a Student object, the compiler is not clever enough to know it.
- To tell the compiler that o is a Student object, use an explicit casting.
- 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 Objects

- ***Upcasting:***
  - When casting an instance of a subclass to a variable of a superclass
  - It is possible, because an instance of a subclass is always an instance of its superclass.
- ***Downcasting:***
  - When casting an instance of a superclass to a variable of its subclass
  - Explicit casting must be used to confirm your intention to the compiler with the (SubclassName) cast notation.

instanceof Operator

- For the downcasting to be successful, you must make sure that the object to be cast is an instance of the subclass.
- If the superclass object is not an instance of the subclass, a runtime ClassCastException occurs.
- For example, if an object is not an instance of Student, it cannot be cast into a variable of Student.
- Therefore, to ensure that the object is an instance of another object before attempting a casting.
- This can be accomplished by using the **instanceof** operator.

instanceof Operator

- Consider the following code:

```
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());
    ...
}
```

-

Casting Objects

- 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.

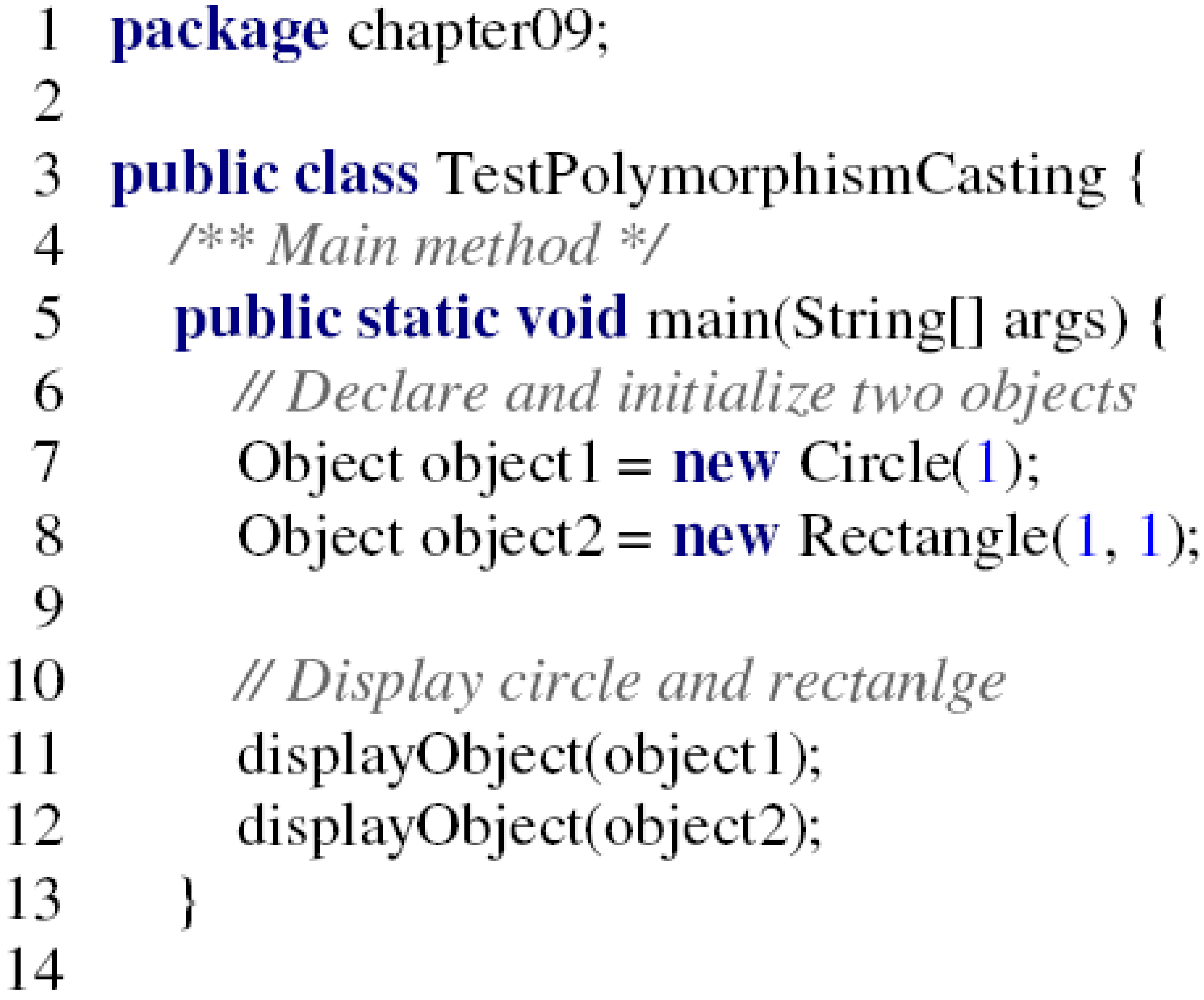
Casting Objects

- why casting is necessary?
- Variable myObject is declared Object.
- The declared type decides which method to match at compile time. Using myObject.getDiameter() would cause a compilation error because the Object class does not have the getDiameter method.
- The compiler cannot find a match for myObject.getDiameter().
- It is necessary to cast myObject into the Circle type to tell the compiler that myObject is also an instance of Circle.

Casting Objects

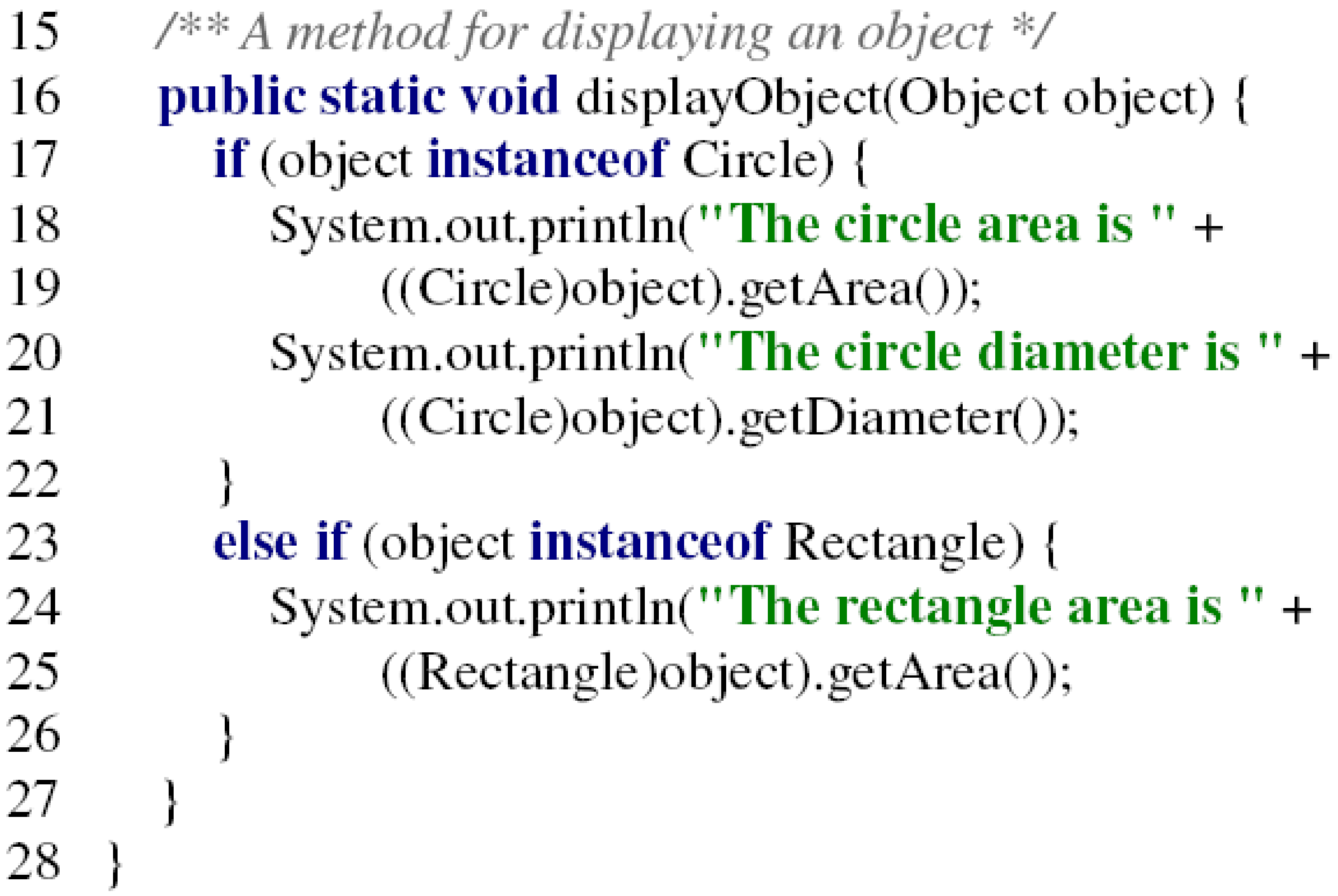
- Why not declare myObject as a Circle type in the first place?
- To enable generic programming, it is a good practice to declare a variable with a supertype, which can accept a value of any subtype.

TestPolymorphismCasting.java



```java
package chapter09;

public class TestPolymorphismCasting {
  /** Main method */
  public static void main(String[] args) {
    // Declare and initialize two objects
    Object object1 = new Circle(1);
    Object object2 = new Rectangle(1, 1);

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

```

TestPolymorphismCasting.java



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

TestPolymorphismCasting.java

- The program uses implicit casting to assign a Circle object to object1 and a Rectangle object to object2, and then invokes the displayObject method to display the information on these objects.
- Casting can only be done when the source object is an instance of the target class.
- The program uses the instanceof operator to ensure that the source object is an instance of the target class before performing a casting

TestPolymorphismCasting.java

- The object member access operator (.) precedes the casting operator.
- Use parentheses to ensure that casting is done before the . operator, as in

```
((Circle)object).getArea();
```

The ArrayList Class

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.
- ArrayList is a class of java.util.

Some methods in ArrayList

- +ArrayList()
  - Creates an empty list.
- +add(o: Object) : void
  - Appends a new element o at the end of this list.
- +add(index: int, o: Object) : void
  - Adds a new element o at the specified index in this list.
- +clear(): void
  - Removes all the elements from this list.
- +contains(o: Object): boolean
  - Returns true if this list contains the element o.

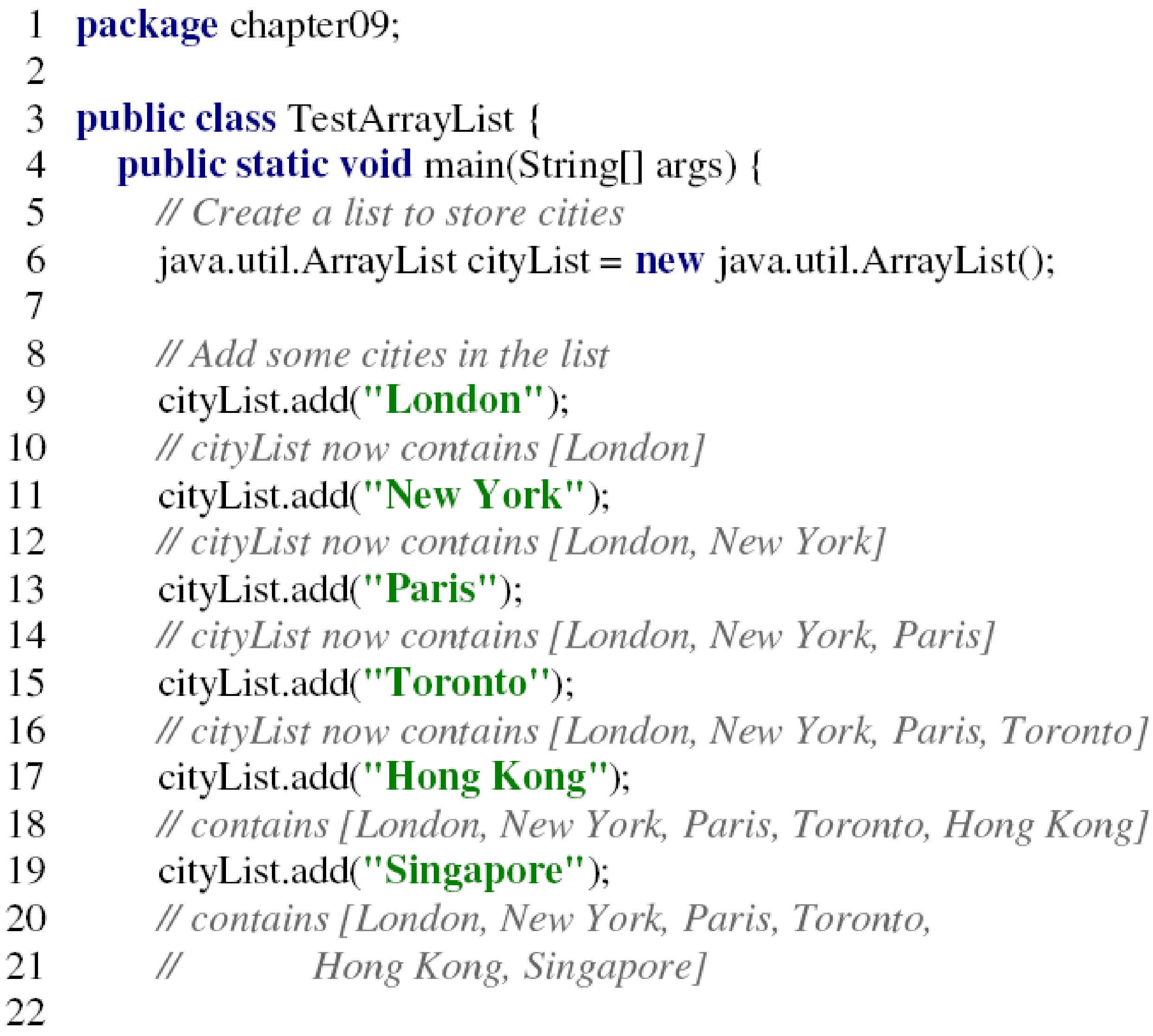
Some methods in ArrayList

- +get(index: int) : Object
  - Returns the element from this list at the specified index.
- +indexOf(o: Object) : int
  - Returns the index of the first matching element in this list.
- +isEmpty(): boolean
  - Returns true if this list contains no elements.
- +lastIndexOf(o: Object) : int
  - Returns the index of the last matching element in this list.

Some methods in ArrayList

- +remove(o: Object): boolean
  - Removes the element o from this list.
- +size(): int
  - Returns the number of elements in this list.
- +remove(index: int) : Object
  - Removes the element at the specified index.
- +set(index: int, o: Object) : Object
  - Sets the element at the specified index.

TestArrayList.java

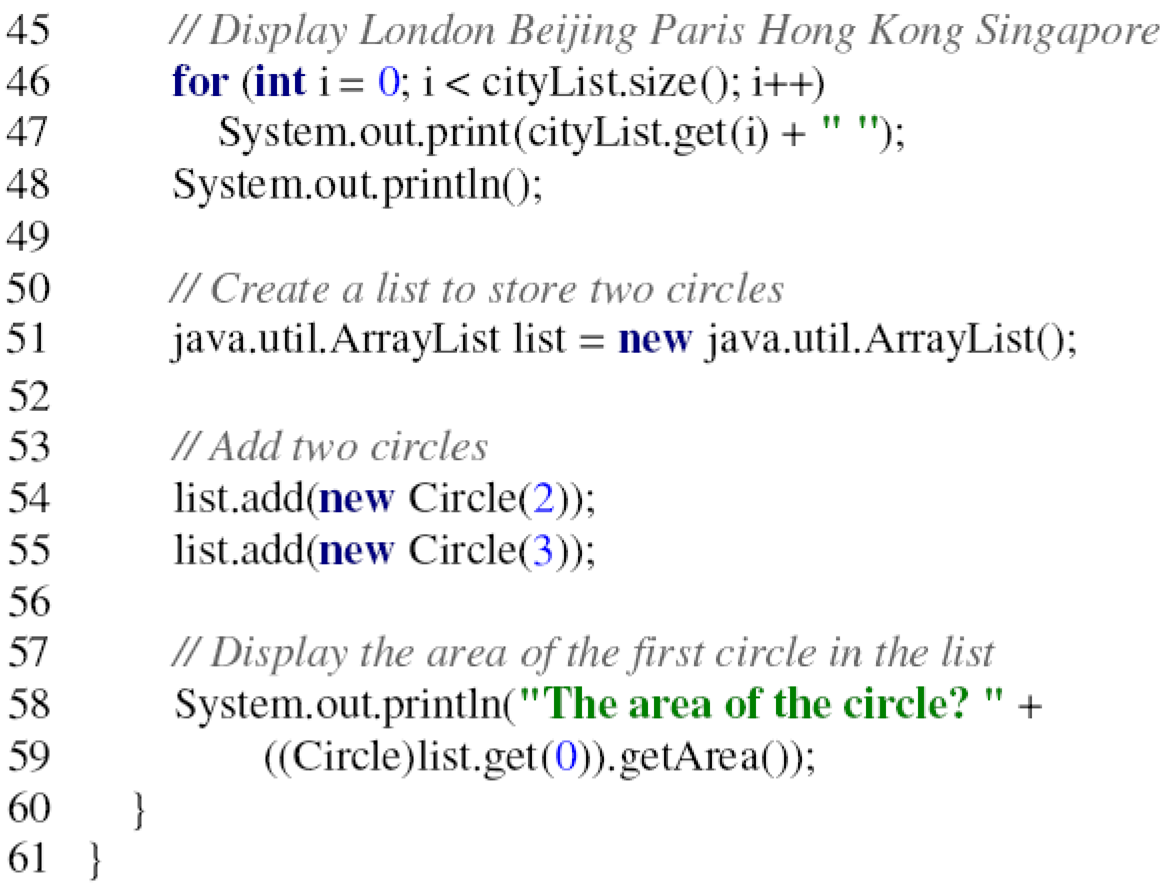


```java
package chapter09;

public class TestArrayList {
  public static void main(String[] args) {
    // Create a list to store cities
    java.util.ArrayList cityList = new java.util.ArrayList();

    // Add some cities in the list
    cityList.add("London");
    // cityList now contains [London]
    cityList.add("New York");
    // cityList now contains [London, New York]
    cityList.add("Paris");
    // cityList now contains [London, New York, Paris]
    cityList.add("Toronto");
    // cityList now contains [London, New York, Paris, Toronto]
    cityList.add("Hong Kong");
    // contains [London, New York, Paris, Toronto, Hong Kong]
    cityList.add("Singapore");
    // contains [London, New York, Paris, Toronto,
    //           Hong Kong, Singapore]

```

TestArrayList.java

```
System.out.println("List size? " + cityList.size());
System.out.println("Is Toronto in the list? " +
    cityList.contains("Toronto"));
System.out.println("The location of New York in the list? '
    + cityList.indexOf("New York"));
System.out.println("Is the list empty? " +
    cityList.isEmpty()); // Print false

// Insert a new city at index 2
cityList.add(2, "Beijing");
// contains [London, New York, Beijing, Paris, Toronto,
//           Hong Kong, Singapore]

// Remove a city from the list
cityList.remove("Toronto");
// contains [London, New York, Beijing, Paris,
//           Hong Kong, Singapore]

// Remove a city at index 1
cityList.remove(1);
// contains [London, Beijing, Paris, Hong Kong, Singapore]
```

TestArrayList.java



```
        // Display London Beijing Paris Hong Kong Singapore
        for (int i = 0; i < cityList.size(); i++)
          System.out.print(cityList.get(i) + " ");
        System.out.println();

        // Create a list to store two circles
        java.util.ArrayList list = new java.util.ArrayList();

        // Add two circles
        list.add(new Circle(2));
        list.add(new Circle(3));

        // Display the area of the first circle in the list
        System.out.println("The area of the circle? " +
          ((Circle)list.get(0)).getArea());
    }
}
```

TestArrayList.java

- You will get a compilation warning “unchecked operation” Ignore it.

- The output:

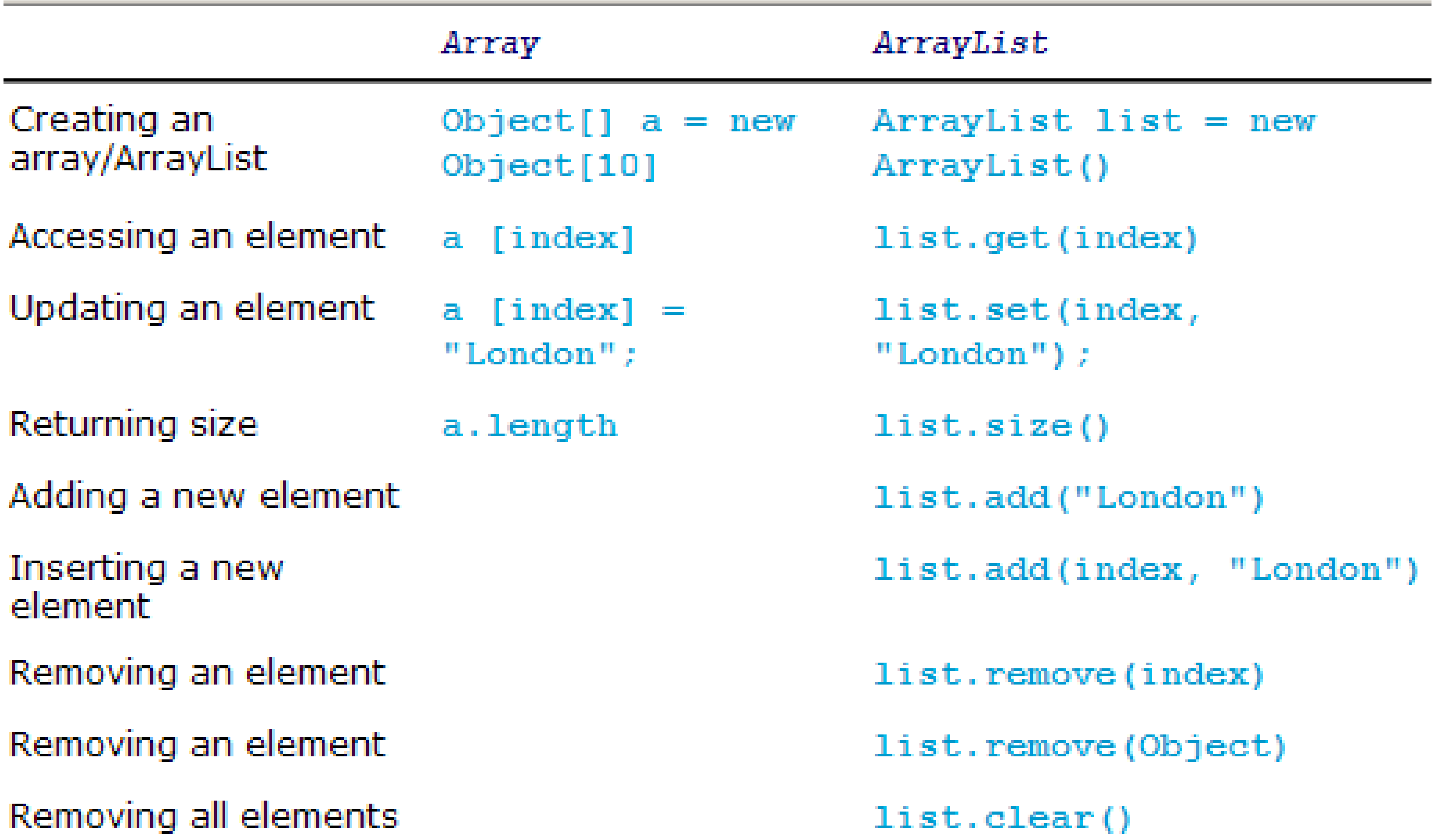
```
List size? 6
Is Toronto in the list? true
The location of New York in the list? 1
Is the list empty? false
London Beijing Paris Hong Kong Singapore
The area of the circle? 12.566370614359172
```

The ArrayList Class

- Differences and Similarity between Arrays and ArrayList:
  - Once an array is created, its size is fixed.
  - You can access an array element using the square bracket notation (e.g., a[index]).
  - When an ArrayList is created, its size is 0.
  - You cannot use the get and set method if the element is not in the list.
  - It is easy to add, insert, and remove elements in a list, but it is rather complex to add, insert, and remove elements in an array.

The ArrayList Class

- Differences and Similarity between Arrays and ArrayList:



	Array	ArrayList
Creating an array/ArrayList	`Object[] a = new Object[10]`	`ArrayList 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()`

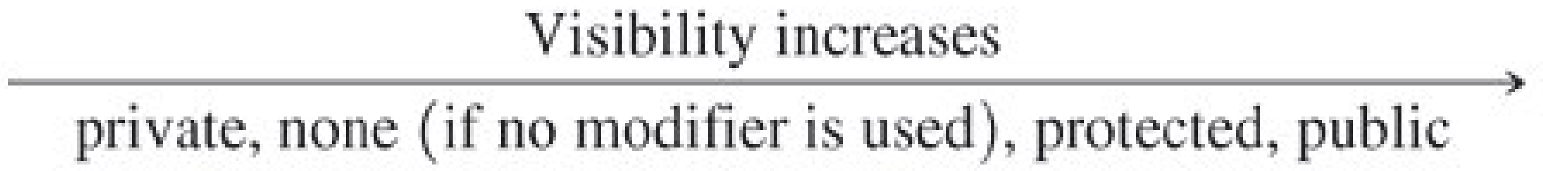
The **protected** Data and Methods

The protected Data and Methods

- 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.
- The modifiers private, protected, and public are known as visibility or accessibility modifiers because they specify how class and class members are accessed.

Visibility modifiers

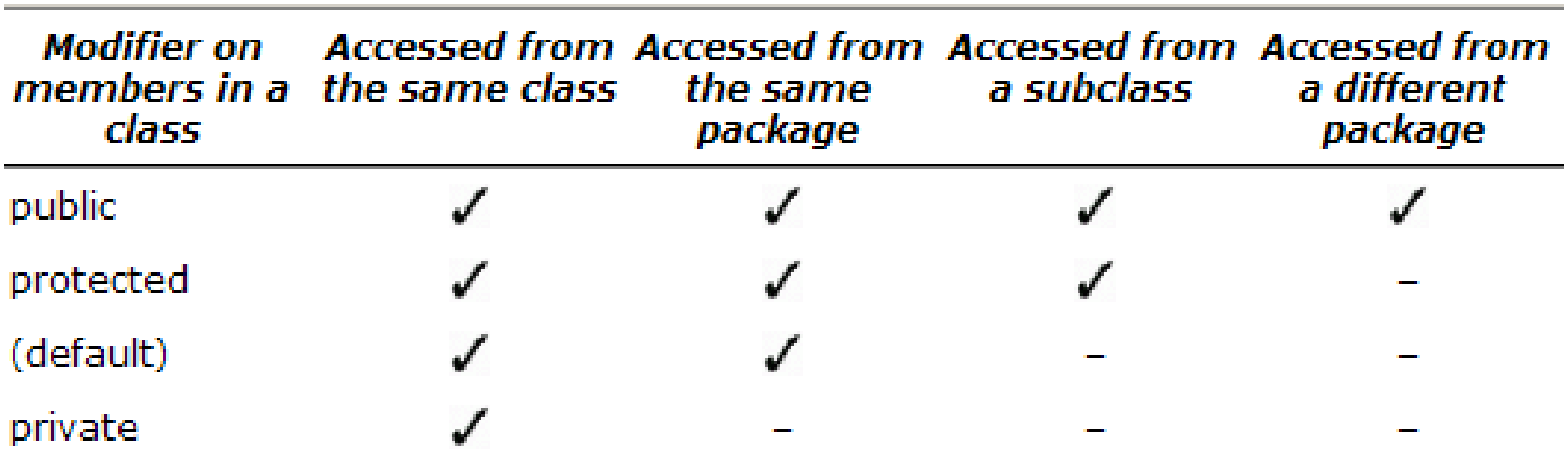
- The visibility of these modifiers increases in this order:



Visibility increases

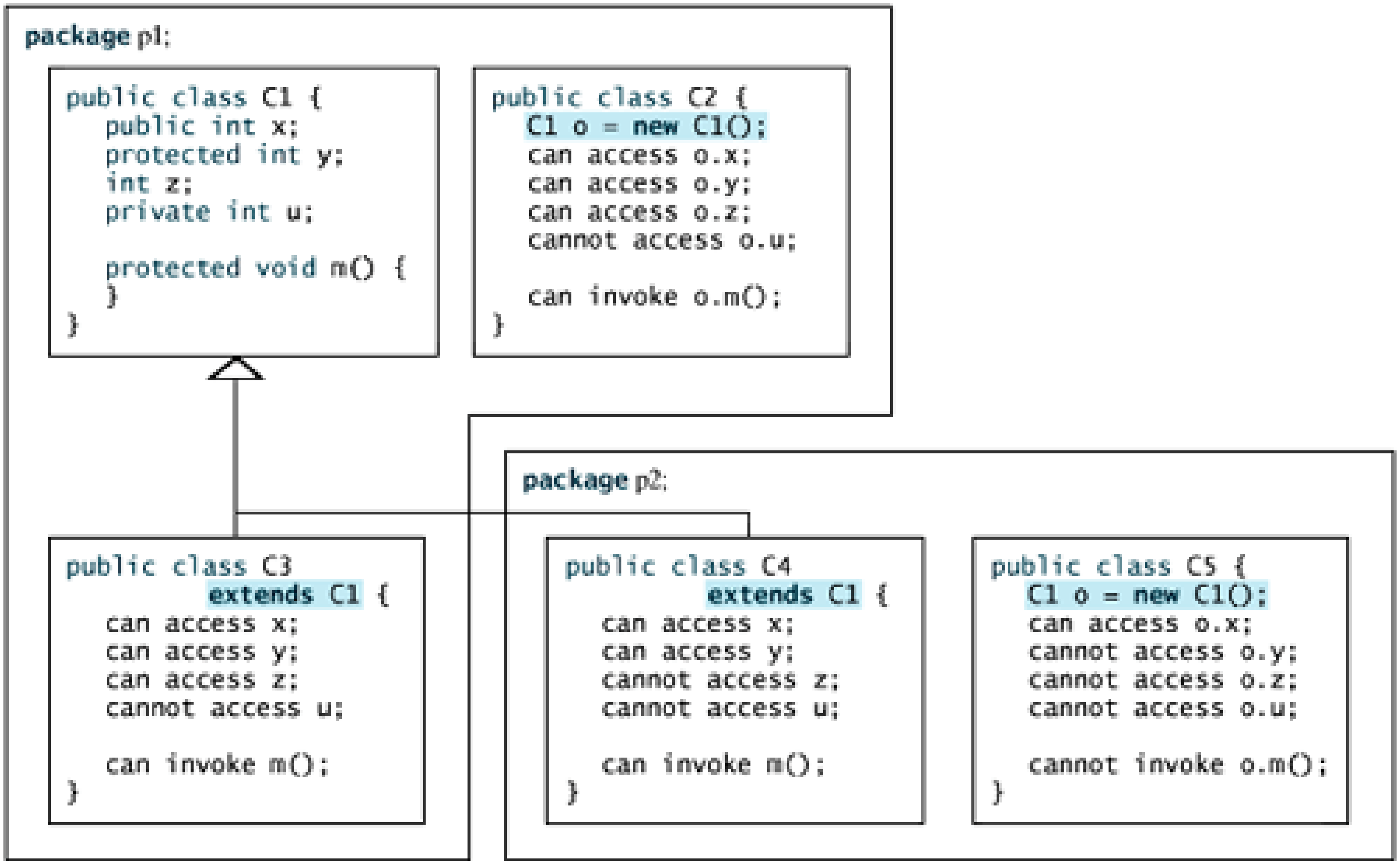
private, none (if no modifier is used), protected, public

- Summarizing the accessibility of the members in a class



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.

The final Classes, Methods, and Variables

The final Classes, Methods, and Variables

- 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.

The final Classes, Methods, and Variables

- 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.

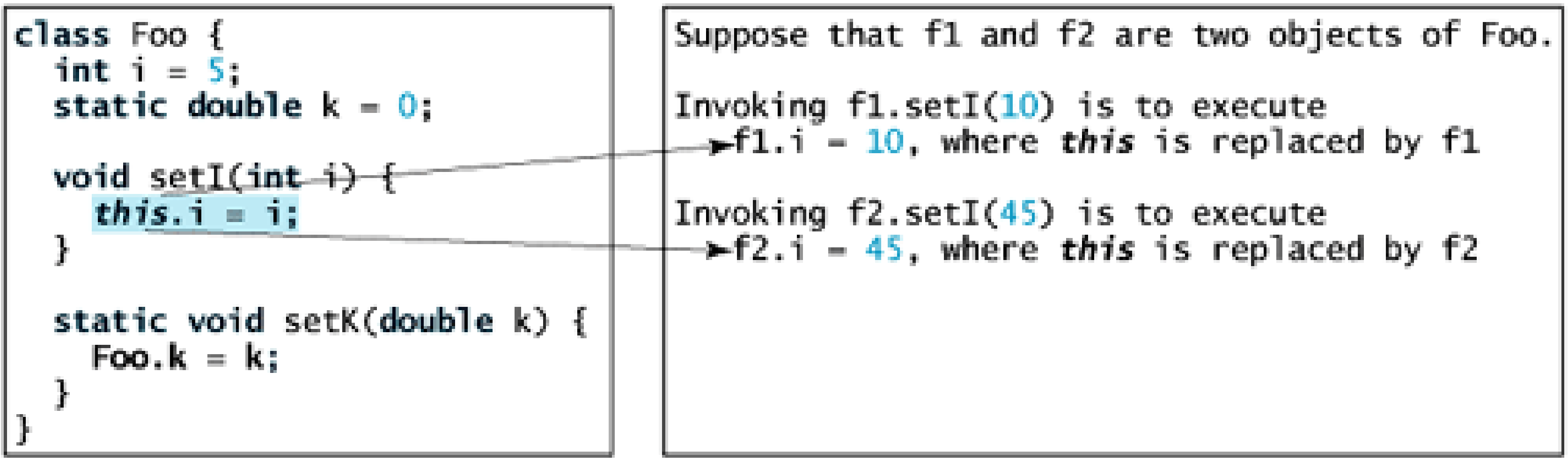
The this Keyword

The this Keyword

- A property (data field) name is often used as the parameter name in a set method for the property.
- In this case, you need to reference the hidden property name in the method in order to set a new value to it.
- A hidden static variable can be accessed simply by using the ClassName.StaticVariable reference.
- A hidden instance variable can be accessed by using the keyword **this**.

The this Keyword

- The keyword this serves as the proxy for the object that invokes the method.



```
class Foo {
  int i = 5;
  static double k = 0;

  void setI(int i) {
    this.i = i;
  }

  static void setK(double k) {
    Foo.k = k;
  }
}
```

(a)

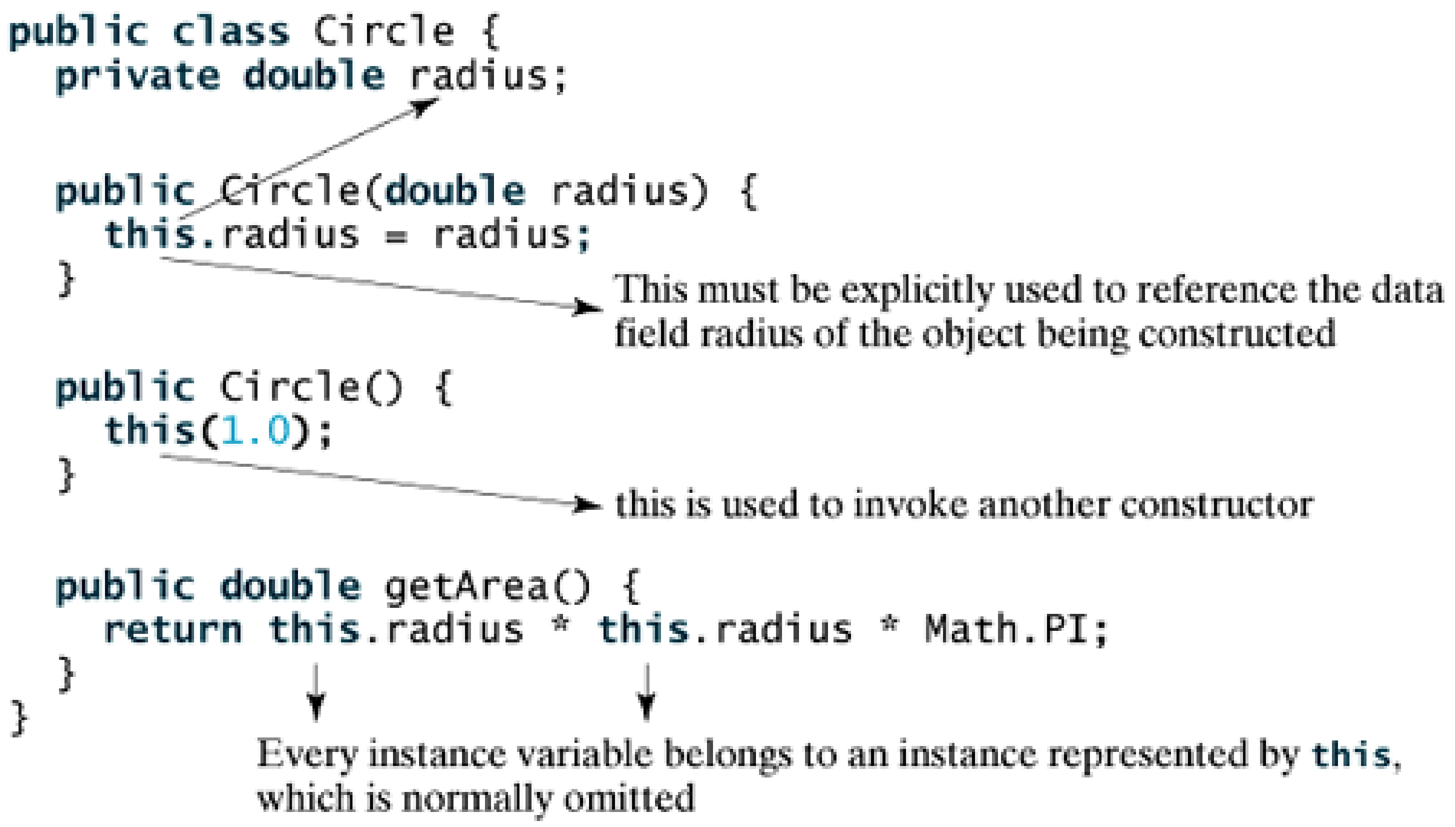
```
Suppose that f1 and f2 are two objects of Foo.

Invoking f1.setI(10) is to execute
  f1.i = 10, where this is replaced by f1

Invoking f2.setI(45) is to execute
  f2.i = 45, where this is replaced by f2
```

(b)

- The line this.i = i means "assign the value of parameter i to the data field i of the calling object."

The this Keyword

- The keyword this can also be used inside a constructor to invoke another constructor of the same class.



```
public class Circle {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  public double getArea() {
    return this.radius * this.radius * Math.PI;
  }
}
```

This must be explicitly used to reference the data field radius of the object being constructed

this is used to invoke another constructor

Every instance variable belongs to an instance represented by this, which is normally omitted

Getting Input from the Console

Getting Input from the Console

- You can obtain input from an input dialog box using the JOptionPane.showInputDialog method.
- Alternatively, you may obtain input from the console.
- Java uses System.out to refer to the standard output device, and System.in to the standard input device.
- By default the output device is the console, and the input device is the keyboard.

Getting Input from the Console

- To perform console output, you simply use the println method to display a primitive value or a string to the console.
- Console input is not directly supported in Java, but you can use the Scanner class to create an object to read input from System.in, as follows:

```
Scanner scanner = new Scanner(System.in);
```

Getting Input from the Console

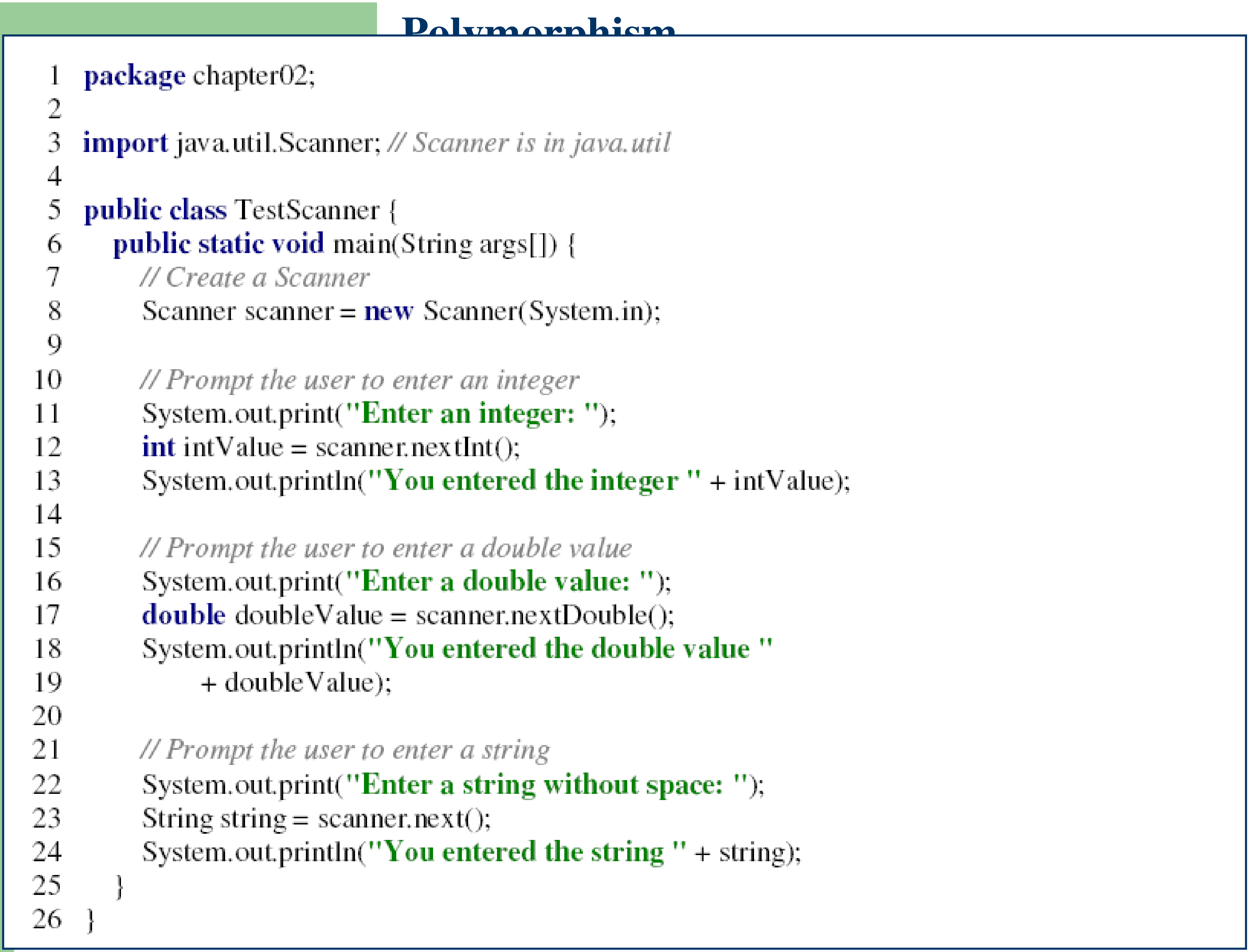
- A Scanner object contains the following methods for reading an input:
  - next(): reading a string. A string is delimited by spaces.
  - nextByte(): reading an integer of the byte type.
  - nextShort(): reading an integer of the short type.
  - nextInt(): reading an integer of the int type.
  - nextLong(): reading an integer of the long type.
  - nextFloat(): reading a number of the float type.
  - nextDouble(): reading a number of the double type.

Getting Input from the Console

- For example, the following statements prompt the user to enter a double value from the console.

```
System.out.print("Enter a double value: ");
Scanner scanner = new Scanner(System.in);
double d = scanner.nextDouble();
```

Polymorphism



```java
package chapter02;

import java.util.Scanner; // Scanner is in java.util

public class TestScanner {
  public static void main(String args[]) {
    // Create a Scanner
    Scanner scanner = new Scanner(System.in);

    // Prompt the user to enter an integer
    System.out.print("Enter an integer: ");
    int intValue = scanner.nextInt();
    System.out.println("You entered the integer " + intValue);

    // Prompt the user to enter a double value
    System.out.print("Enter a double value: ");
    double doubleValue = scanner.nextDouble();
    System.out.println("You entered the double value "
        + doubleValue);

    // Prompt the user to enter a string
    System.out.print("Enter a string without space: ");
    String string = scanner.next();
    System.out.println("You entered the string " + string);
  }
}
```

References

References

- Y. Daniel Liang, **Introduction to Java Programming**, Sixth Edition, Pearson Education, 2007. (Chapter 9)

The End