Lecture-16

By

Dr. Bharati Mishra

Objectives

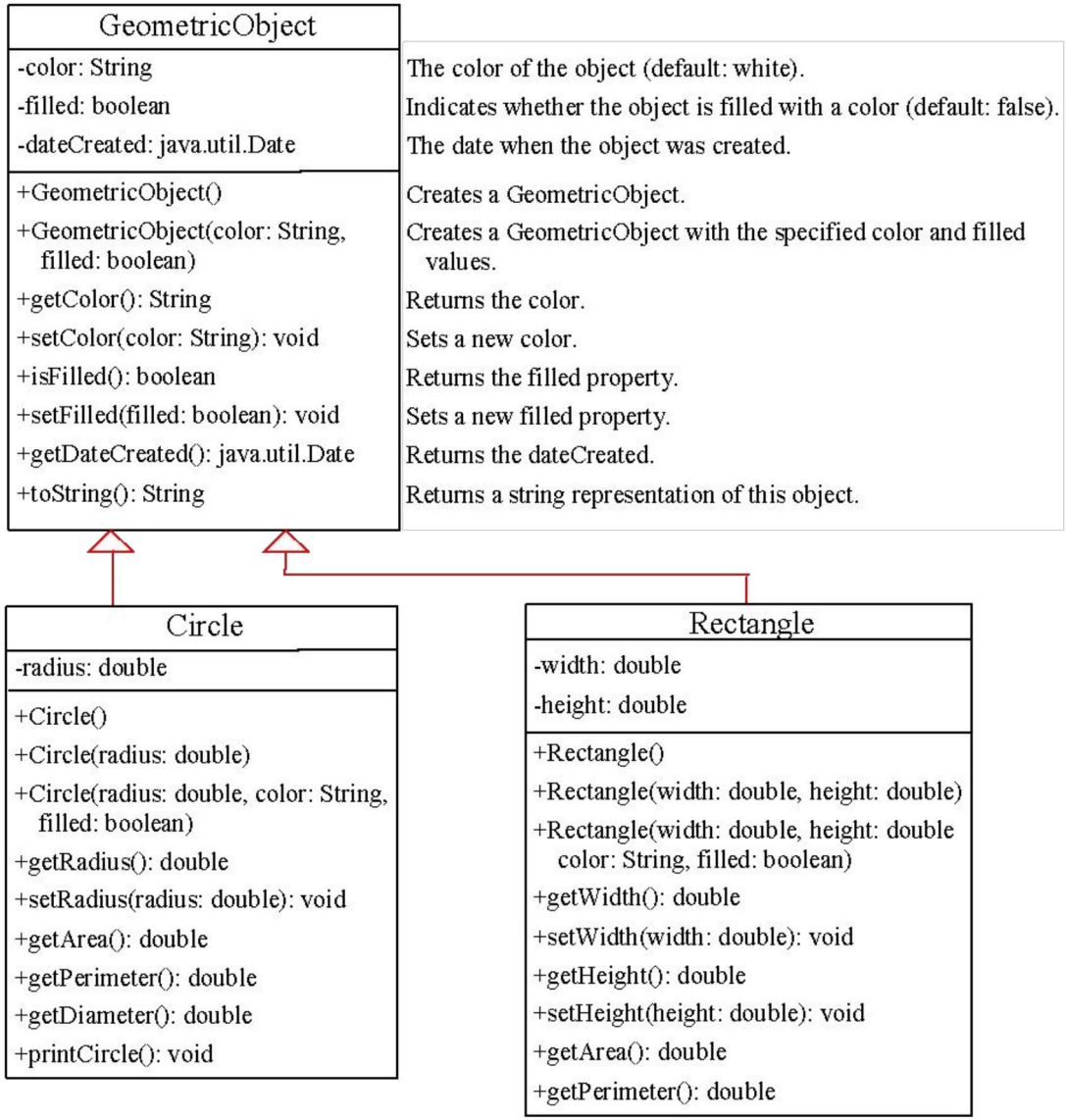
- Chapter 11
  - Inheritance
    - Subclass, superclass
    - **super** keyword
    - **abstract** keyword
    - Overriding methods
    - Polymorphism and dynamic binding
    - **final** keyword
  - ArrayList

Motivation

- Suppose you want to 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.

Inheritance

Superclass



GeometricObject1

Circle4

Rectangle1

TestCircleRectangle

Run

Inheritance

- Models “is-a” relationship
- Not all “is-a” relationships should be modeled using inheritance
  - For class *A* to extend class *B*, *A* should contain more detailed information than *B*.
- Do not blindly extend a class just for the sake of reusing methods!

Inheritance

- A subclass does not inherit the private members of its parent class.
- A subclass is not a subset of the superclass
  - Contains more information!
- Java does not support multiple inheritance
  - It can be achieved through interfaces (an advanced topic)

abstract

- An *abstract class* may or may not include abstract methods.
- Abstract classes cannot be instantiated, but they can be sub-classed.
- An *abstract method* is a method that is declared without an implementation.
- If a class includes abstract methods, the class itself *must* be declared abstract

abstract

- An *abstract class* example

```
public abstract class GraphicObject
{
  // declare fields
  // declare non-abstract methods
  abstract void draw();
}
```

Inheritance & Constructor

Constructors

- Are superclass constructors inherited?
  - No. Unlike properties and methods, a superclass's constructors are not inherited in the subclass.

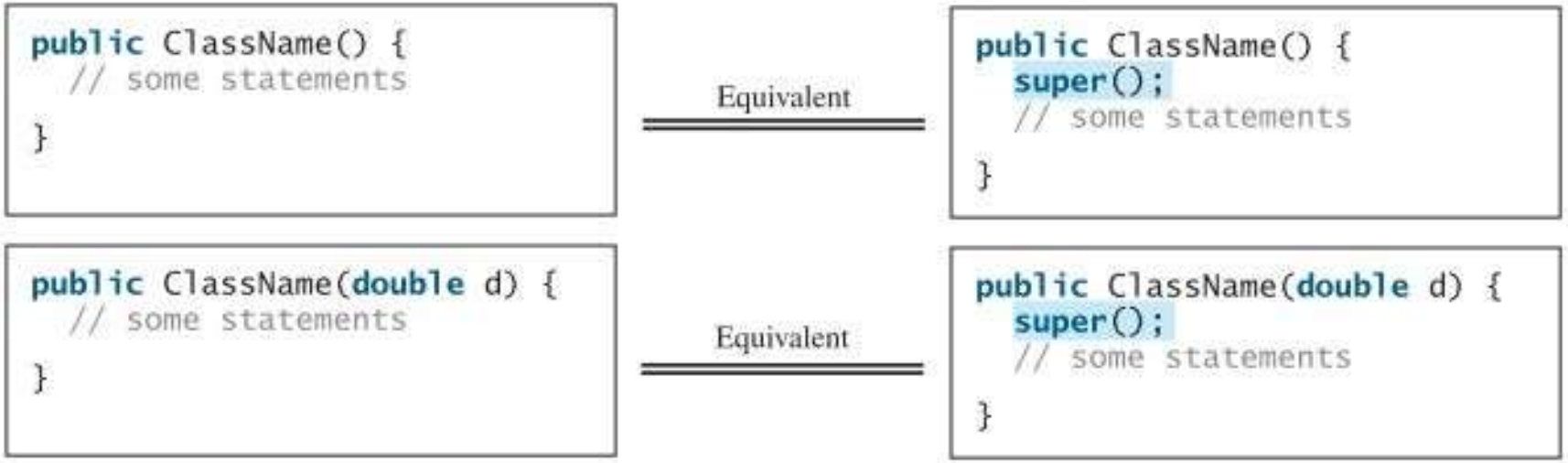
Constructors

1. The constructors are invoked either explicitly:
   - Using the **super** keyword
2. Or implicitly:
   - If the keyword **super** is not explicitly used, the superclass's no-arg constructor is automatically invoked.

Constructors

- 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.

- Example


public ClassName() {
// some statements
}
Equivalent
public ClassName() {
super();
// some statements
}
public ClassName(double d) {
// some statements
}
Equivalent
public ClassName(double d) {
super();
// some statements
}

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

Constructors

- 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.

Chaining

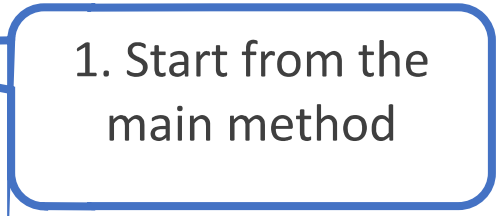
- Constructor chaining
  - Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain.
  - This is called *constructor chaining*.

Chaining

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Performs Faculty's tasks");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Performs Employee's tasks ");
  }

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

class Person {
  public Person() {
    System.out.println("(1) Performs Person's tasks");
  }
}
```

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 invoked");
  }
}
```



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 invoked");
  }
}
```

2. Invoke Faculty 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 invoked");
  }
}
```

3. Invoke Employee's no-arg 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 invoked");
  }
}
```

4. Invoke Employee(String)

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 invoked");
  }
}
```

5. Invoke Person() 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");
  }
}
```

6. Execute println

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 invoked");
  }
}
```

7. Execute println

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 invoked");
  }
}
```

8. Execute println

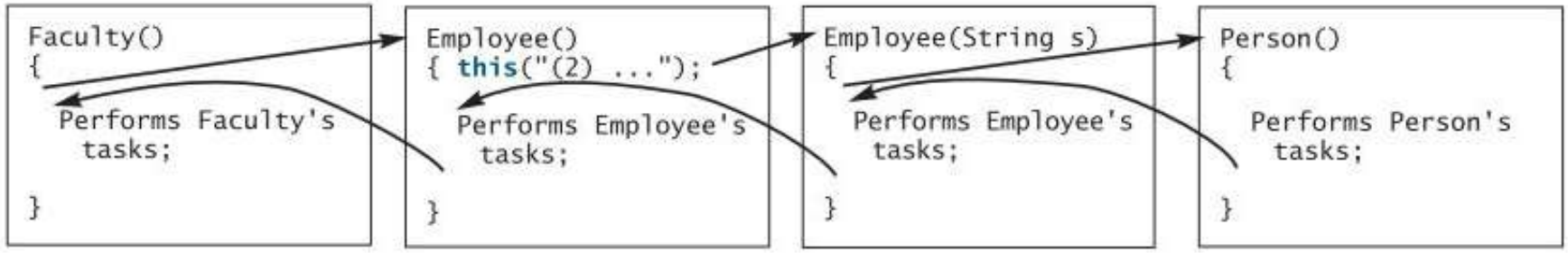
Chaining

```
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 invoked");
  }
}
```

9. Execute println

Chaining

- Chaining in the program:



Chaining

- 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");
  }
}
```

Subclass/Superclass

Subclass

- A subclass extends properties and methods from the superclass.
- You can also:
  - Add new properties
  - Add new methods
  - Override the methods of the superclass

Subclass

- 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

Overriding

- 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*.

Overriding

- Method overriding

```
public class Circle4 extends GeometricObject1 {
  // Other methods are omitted

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

Overriding

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

Overriding

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

Overriding

- Overriding vs. overloading

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

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

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

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

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

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

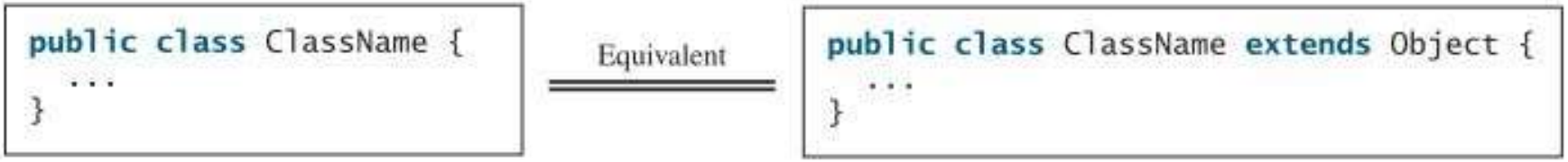
Object Class

Object Class

- 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.

Object Class

- Equivalent



Object Class

- The toString() method returns a string representation of the object.
- The default implementation returns a string consisting of:
  - a class name of which the object is an instance,
  - (@),
  - 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

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

Polymorphism

- Three pillars of OOP
  1. Encapsulation
  2. Inheritance
  3. Polymorphism

Code

- Polymorphism

```
public class PolymorphismDemo {
  /** Main method */
  public static void main(String[] args) {
    // Display circle and rectangle properties
    displayObject(new Circle4(1, "red", false));
    displayObject(new Rectangle1(1, 1, "black", true));
  }

  /** Display geometric object properties */
  public static void displayObject(GeometricObject1 object) {
    System.out.println("Created on " + object.getDateCreated() +
      ". Color is " + object.getColor());
  }
}
```

- If polymorphism is used:
  - Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime.
  - This capability is known as *dynamic binding*.

Code

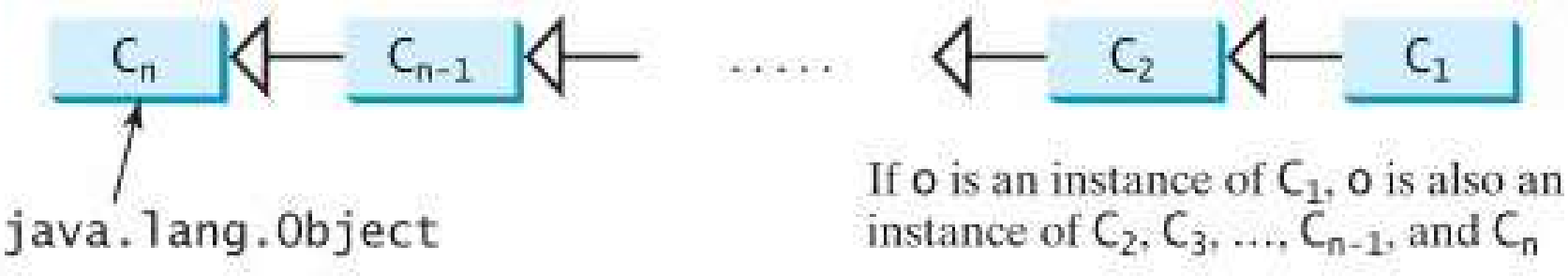
```
public class DynamicBindingDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

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

class GraduateStudent extends Student {
}

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

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

Binding

- 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.

Binding

- 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.



Binding

- Matching a method signature
  - The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time.
- Binding a method
  - A method may be implemented in several subclasses.
  - The Java Virtual Machine dynamically binds the implementation of the method at runtime.

- Polymorphism allows methods to be used generically for a wide range of object arguments.
- This is known as generic programming.

Binding

- 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

Casting

- Casting can be used to convert an object of one class type to another within an inheritance hierarchy.

```
Object o = new Student(); // Implicit casting
m(o);
```

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

Casting

- The following statement causes a compilation error:

```
Student b = o;
```

- Because a Student object is always an instance of Object, but an Object is not necessarily an instance of Student!

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

Casting

- Explicit casting must be used when casting an object from a superclass to a subclass.
- This type of casting may not always succeed.

instanceof

- Use the **`instanceof`** operator to test whether an object is an instance of a class.

```
Object myObject = new Circle();
... // Some lines of code
```

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

Code

- This example creates two geometric objects: a circle, and a rectangle. It displays the area and diameter if the object is a circle, and displays area if the object is a rectangle.

TestPolymorphismCasting

Run

Equality

equals()

- 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);
}
```

equals()

- You can override **`equals()`** method in your class

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

ArrayList

ArrayList

- 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

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

Program

- ArrayList
- You will get a compilation warning "unchecked operation." Ignore it. This warning can be fixed using generic types in Chapter 20.

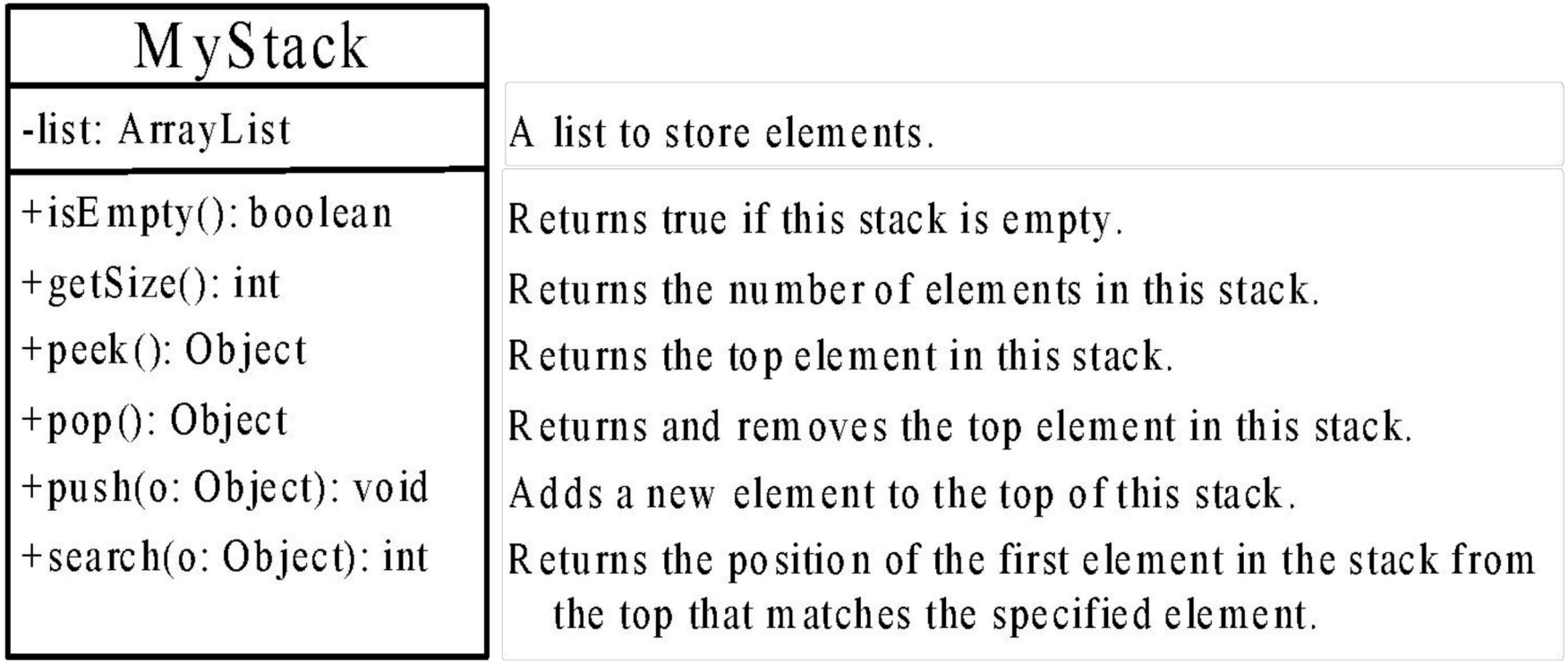
TestArrayList

Run

Program

- MyStack

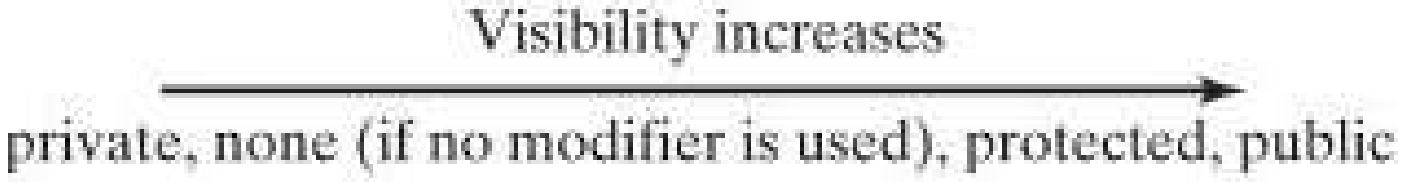
MyStack



Accessibility

protected

- Another visibility modifier
- 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.



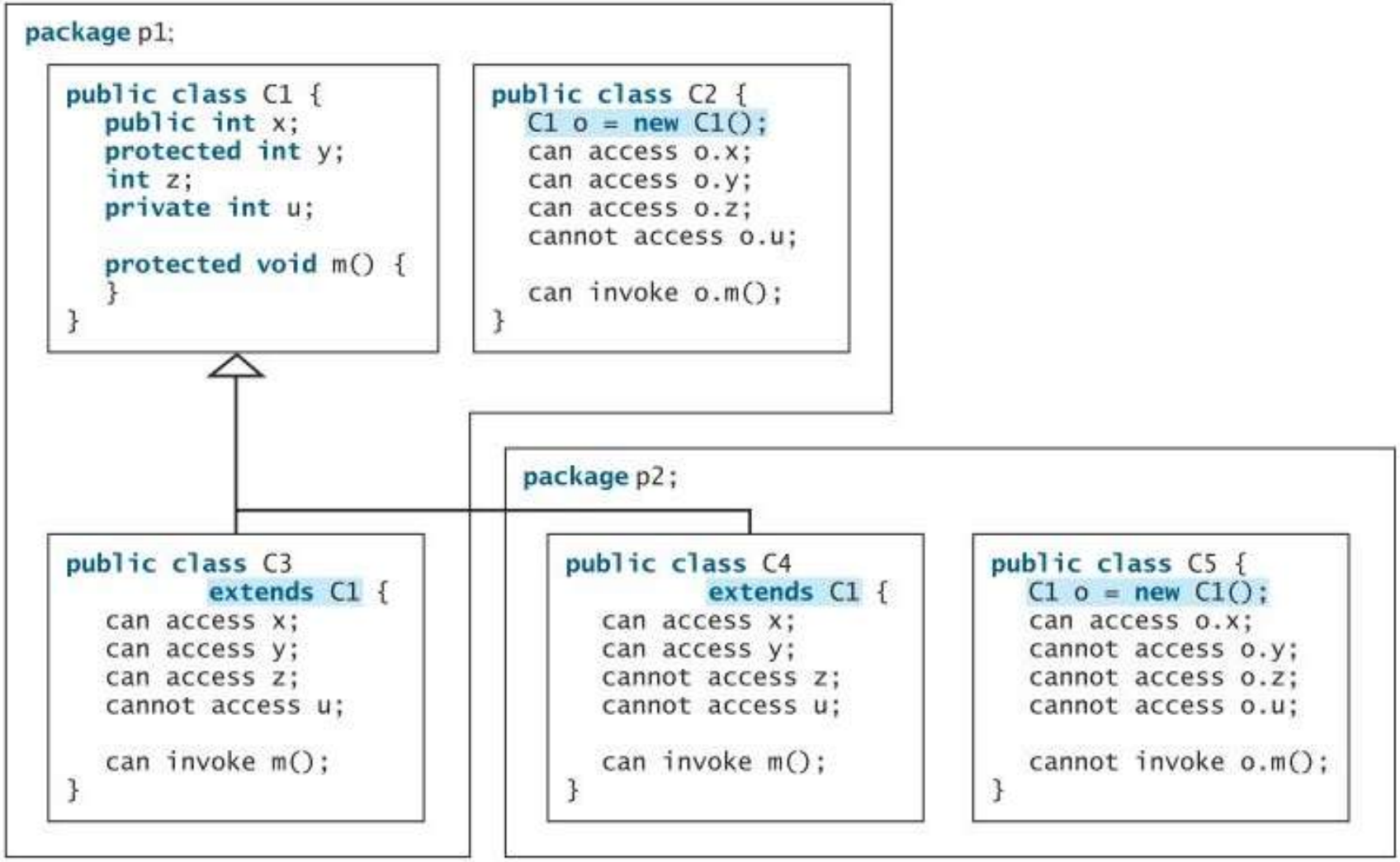
Accessibility

- 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	✓	–	–	–

Accessibility

- Accessibility example



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.

final keyword

final

- A **`final`** class cannot be extended

```
public final class C {
  // Data fields, constructors, and methods omitted
}
```

- A **`final`** variable is a constant

- A **`final`** method cannot be overridden by its subclasses.

```
public class Test {
  // Data fields, constructors, and methods omitted

  public final void m() {
    // Do something
  }
}
```