Chapter 11 Inheritance and Polymorphism

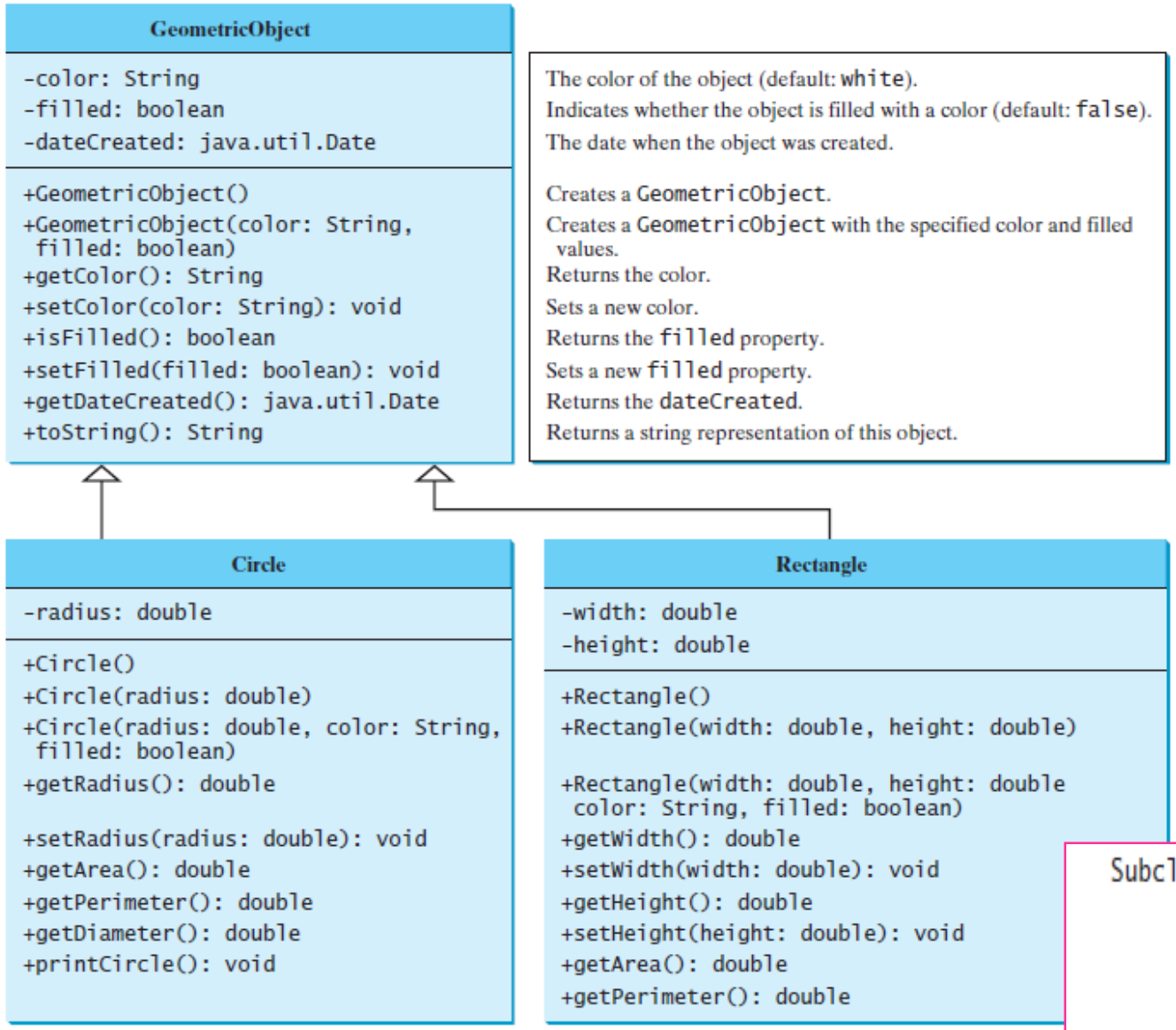
Motivations

Some classes have many common features.

How to avoid redundancy? The answer is to use:

Inheritance: derive new classes from existing classes

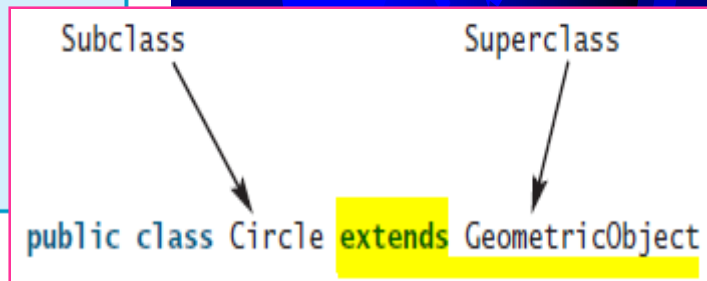
Superclasses and Subclasses



The GeometricObject class is the superclass for Circle and Rectangle.

- **Superclass**:
  parent class
  base class
- **Subclass:**
  child class,
  extended class,
  derived class

- A subclass
  - **inherits accessible members** (data/method)
  - may **add new members**



```
public Circle (double radius, String color, boolean filled) {
  this.radius = radius;
  this.color = color; // Illegal
  this.filled = filled; // Illegal
}
```

- Why is it wrong?

This is wrong, because the private data fields `color` and `filled` in the `GeometricObject` class cannot be accessed in any class other than in the `GeometricObject` class itself. The only way to read and modify `color` and `filled` is through their `get` and `set` methods.

- Private data fields cannot be used directly in a subclass.
  - should be used through public method defined in the superclass.

- A subclass should contain more information than its superclass.
  - e.g., a square is a rectangle, but nothing to extend from a rectangle
    - Should **Square** class extend: **GeometricObject** class /**Rectangle** class ?

- Do not blindly extend a class just for the sake of reusing methods.
  - A subclass and its superclass must have the *is-a* relationship.
  - e.g., it makes no sense for a **Tree** class to extend a **Person** class,
    - even though they share common properties such as height and weight.

Using the Keyword `super`

- keyword ***this*** refers to the calling object
- keyword ***super*** refers to the superclass of the class
  - to call a superclass constructor
    - 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.
      - The keyword super appear first in the constructor.
  - to call a superclass method
    - Superclass. StaticMethod()

```
public Circle (double radius, String color, boolean filled) {
  super(color, filled);
  this.radius = radius;
}
```

superclass's Constructor

- A constructor may use the keyword *super* explicitly to invoke its superclass's constructor.
  - *If not explicitly used*, the compiler puts super() as the first statement in the constructor: *automatically invoked*.

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

is equivalent to

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

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

is equivalent to

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

Constructor Chaining

Creating an instance of a class invokes all the superclasses' constructors along the inheritance chain.

```
1 public class Faculty extends Employee {
2   public static void main(String[] args) {
3     new Faculty();
4   }
5
6   public Faculty() {
7     System.out.println("(4) Performs Faculty's tasks");
8   }
9 }
10
11 class Employee extends Person {
12   public Employee() {
13     this("(2) Invoke Employee's overloaded constructor");
14     System.out.println("(3) Performs Employee's tasks ");
15   }
16
17   public Employee(String s) {
18     System.out.println(s);
19   }
20 }
21
22 class Person {
23   public Person() {
24     System.out.println("(1) Performs Person's tasks");
25   }
26 }
```

```
(1) Performs Person's tasks
(2) Invoke Employee's overloaded constructor
(3) Performs Employee's tasks
(4) Performs Faculty's tasks
```

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 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");
  }
}
```

5. 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");
  }
}
```

6. 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");
  }
}
```

7. 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");
  }
}
```

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

Superclass should provide a 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");
  }
}
```

- **Apple**'s default constructor automatically invokes **Fruit**'s no-arg constructor.
- However, **Fruit** does not have a no-arg constructor, because **Fruit** has an explicit constructor defined.
- Therefore, the program cannot be compiled.

Tip: It is better to provide a no-arg constructor for every class to avoid programming errors.

Declaring a 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

Overriding Methods in the Superclass

A subclass inherits methods from a superclass.

Sometimes the subclass needs modify the implementation of a method defined in the superclass. This is called *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

- 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.

- A static method cannot be overridden.
  - If a static method defined in the superclass is redefined in a subclass, to call the method defined in the superclass:
    - SuperClassName.staticMethodName();

Overriding vs. Overloading

(same signature) (same name, but different signature)

```
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);
  }
}
```

10.0

10.0

```
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);
  }
}
```

10

20.0

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 default superclass of the class is Object.

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

Equivalent

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

The toString() method in Object

☞returns a string that describe the object :

"**class name@ object's memory address**"

```
Loan loan = new Loan();
System.out.println( loan.toString() );
```

- The code displays "Loan@15037e5"
- This message is not very helpful or informative.

the same:

System.out.println(loan)
System.out.println(loan.toString())

System.out. println(object)

System.out. print(object)

equivalent to:

System.out. println(object.toString())

System.out. print(object.toString())

The toString() method in Object

```
Loan loan = new Loan();
System.out.println( loan.toString() );
```

- The code displays "Loan@15037e5"
- This message is not very helpful or informative.

Usually you should override the toString method so that it returns a descriptive string. For example,

```
overridden in the GeometricObject class

public String toString() {
  return "created on " + dateCreated + "\ncolor: " + color +
    " and filled: " + filled;
}
```

Polymorphism and Dynamic Binding

```
1 public class DynamicBindingDemo {
2   public static void main(String[] args) {
3     m(new GraduateStudent());
4     m(new Student());
5     m(new Person());
6     m(new Object());
7   }
8
9   public static void m(Object x) {
10    System.out.println(x.toString());
11  }
12 }
13
14 class GraduateStudent extends Student {
15 }
16
17 class Student extends Person {
18   public String toString() {
19     return "Student";
20   }
21 }
22
23 class Person extends Object {
24   public String toString() {
25     return "Person";
26   }
27 }
```

Method m takes a parameter of the Object type.

Polymorphism *(means: many forms).*

x **may be an instance of** the following classes: GraduateStudent, Student, Person, or Object.

* Those classes have their own implementation of the toString() method

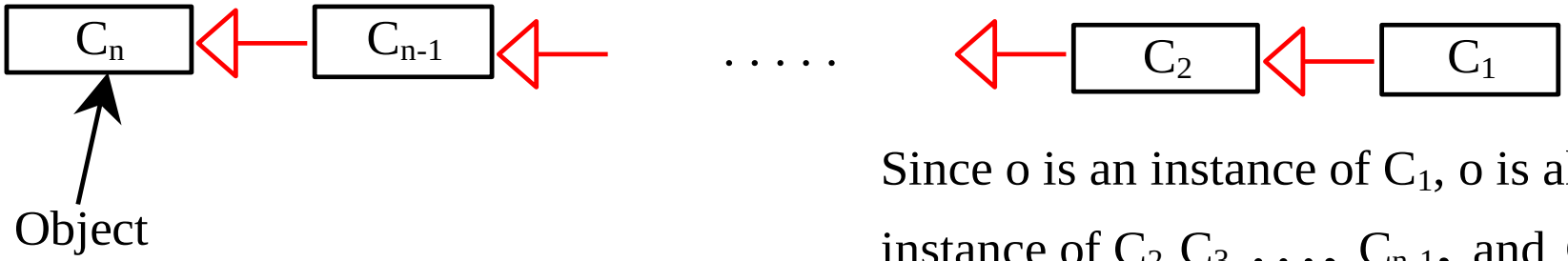
Dynamic binding

☞Which implementation of toString() is used will be determined dynamically by the Java Virtual Machine at runtime.

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

Dynamic Binding

- If o invokes a method p, the JVM searches the implementation for the method p in the order of C_1, C_2, ..., C_{n-1} and C_n, until it is found.
  - Once an implementation is found, the search stops and the first-found implementation is invoked.
  - C_1 is the most specific class.



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

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";
  }
}
```

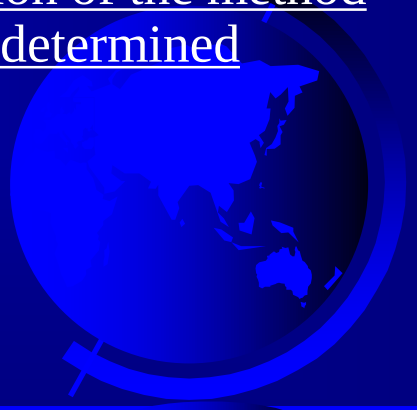
☞Polymorphism is known as generic programming.

one method to be used generically for a wide range of object arguments.

parameter type： superclass (e.g., Object)

argument type： subclass (e.g., Student or String)

– The implementation of the method (e.g., toString) is determined dynamically.



Casting Objects

Type Casting can be used to

- convert **a variable of one primitive type** to another.
- convert **an object of one class type** to another within an inheritance hierarchy.

```
public static void m(Object o){…}

m( new Student() );
```

The second statement is equivalent to:

```
Object o = new Student(); // Implicit casting: from Subclass to Superclass
m(o);
```

Legal, because an instance of Student is automatically an instance of Object.

Explicit Casting from Superclass to Subclass

Object o = new Student();

Student b = o; //compilation error !

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

The `instanceof` Operator

To test whether an object is an instance of a class:

```
Object myObject = new Circle();
...

/** Perform casting if myObject is an instance of Circle */

if (myObject instanceof Circle) {
  System.out.println("The circle diameter is " +
    ( (Circle)myObject).getDiameter());
  ...

}
```

Example: Demonstrating Polymorphism & Casting

- Creates two geometric objects: a circle, and a rectangle.
- The method displayGeometricObject()
  - displays the area and diameter if the object is a circle
  - displays area if the object is a rectangle.

```
 1 public class CastingDemo {
 2   /** Main method */
 3   public static void main(String[] args) {
 4     // Create and initialize two objects
 5     Object object1 = new Circle4(1);
 6     Object object2 = new Rectangle1(1, 1);
 7
 8     // Display circle and rectangle
 9     displayObject(object1);
10     displayObject(object2);
11   }
12
13   /** A method for displaying an object */
14   public static void displayObject(Object object) {
15     if (object instanceof Circle4) {
16       System.out.println("The circle area is " +
17         ((Circle4)object).getArea());
18       System.out.println("The circle diameter is " +
19         ((Circle4)object).getDiameter());
20     }
21     else if (object instanceof Rectangle1) {
22       System.out.println("The rectangle area is " +
23         ((Rectangle1)object).getArea());
24     }
25   }
26 }
```

equals() method of Object class

- Invocation: object1.equals(object2);
- The default implementation :
  - checks whether two reference variables point to the same object

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

- Compares the contents of two objects.
  - Overridden in subclasses: For example, in the Circle class.

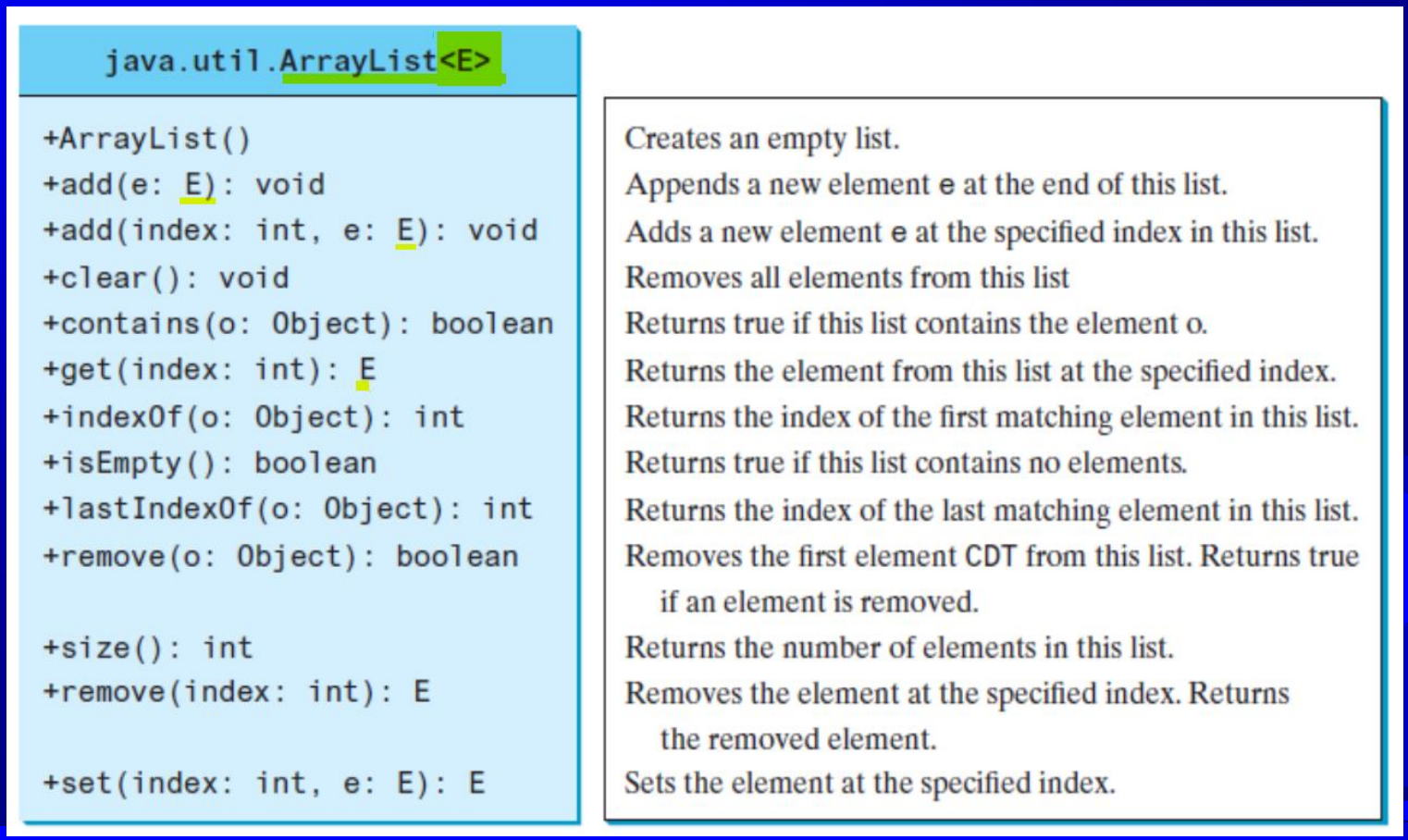
```
public boolean equals(Object o) {
  if (o instanceof Circle) {
    return radius == ((Circle)o).radius;
  }
  else
    return false;
}
```

ArrayList Class

An array's size is fixed once the array is created.

Java ArrayList class can store an unlimited number of objects.

ArrayList is known as a generic class with a generic type E.



Generic Type

ArrayList<E> : a generic class with a generic type E.

You can specify a concrete type to replace E

Since JDK 7, the statement

```
ArrayList <AConcreteType> list = new ArrayList<AConcreteType>();
```

can be simplified by

```
ArrayList<AConcreteType> list = new ArrayList<>();
```

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

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

```java
import java.util.ArrayList;

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

    // Add some cities in the list
    cityList.add("London");
    // cityList now contains [London]
    cityList.add("Denver");
    // cityList now contains [London, Denver]
    cityList.add("Paris");
    // cityList now contains [London, Denver, Paris]
    cityList.add("Miami");
    // cityList now contains [London, Denver, Paris, Miami]
    cityList.add("Seoul");
    // Contains [London, Denver, Paris, Miami, Seoul]
    cityList.add("Tokyo");
    // Contains [London, Denver, Paris, Miami, Seoul, Tokyo]

    System.out.println("List size? " + cityList.size());
    System.out.println("Is Miami in the list? " +
      cityList.contains("Miami"));
    System.out.println("The location of Denver in the list? "
      + cityList.indexOf("Denver"));
    System.out.println("Is the list empty? " +
      cityList.isEmpty()); // Print false

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

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

```
List size? 6
Is Miami in the list? true
The location of Denver in the list? 1
Is the list empty? false
[London, Xian, Paris, Seoul, Tokyo]
```

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();`

Array Lists from/to Arrays

- an array => an ArrayList

```
String[] array = {"red", "green", "blue"};

ArrayList<String> list = new ArrayList<>(Arrays.asList(array));
```

- … = new ArrayList<>(Arrays.asList("red", "green", "blue"));
- import java.util.Arrays;

Array Lists from/to Arrays

- an array => an ArrayList

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

- an ArrayList => an array

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

max and min in an Array List

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

```
String[] array = {"red", "green", "blue"};
System.out.println(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);
```

use **ArrayList** to implement MyStack Classes

A stack to hold objects.

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

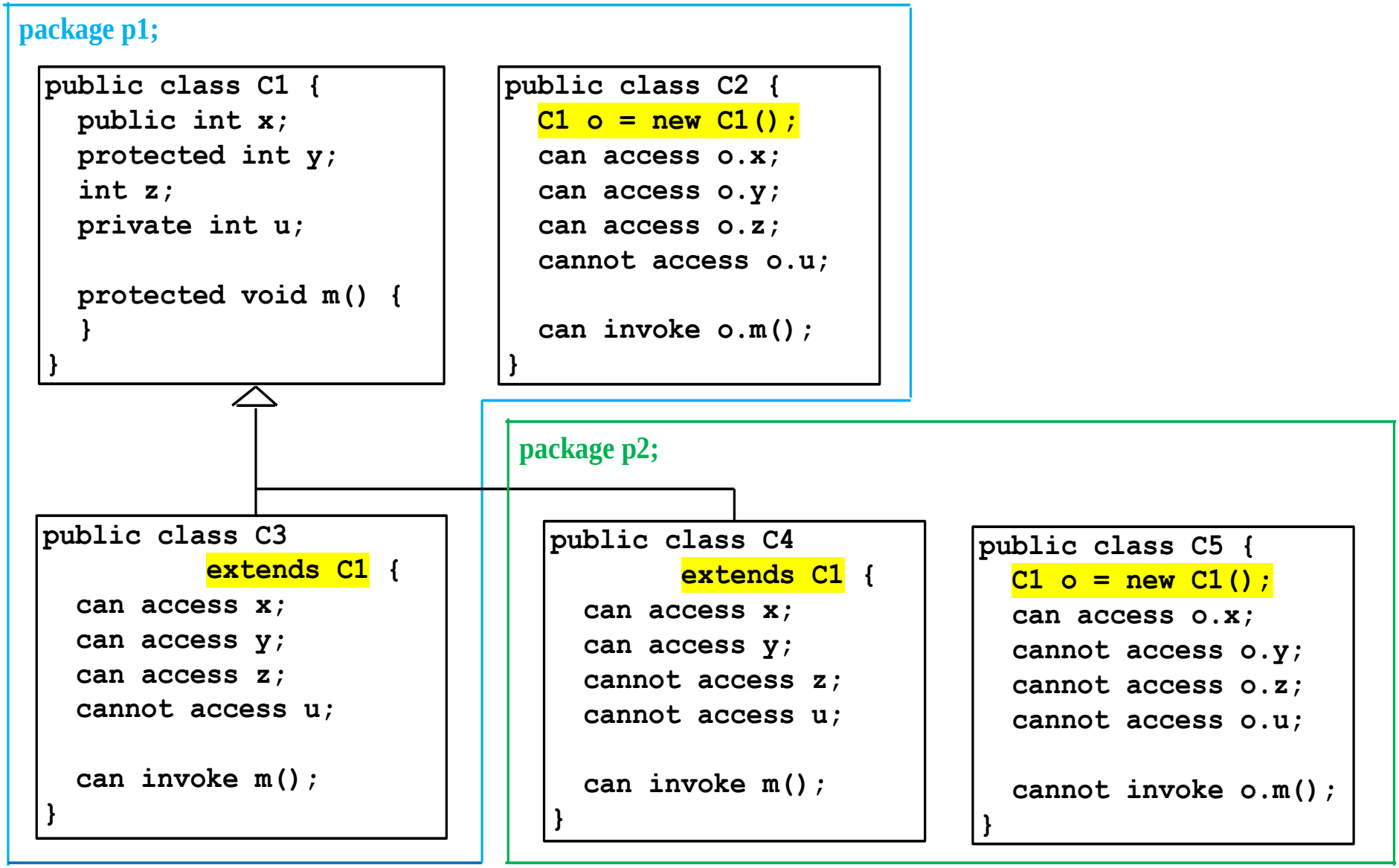
- A **`protected`** data/method in a public class can be accessed by any class
  - in the same package
  - Or in its **subclasses**, even if the subclasses are in a different package.

- private, default, protected, public

Accessibility Summary

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

Visibility Modifiers



A Subclass Cannot Weaken the Accessibility

- A subclass may override a protected method in its superclass and change its visibility to public.
- However, a subclass cannot weaken the accessibility of a method defined in the superclass.

- For example,
  - if a method is defined as public in the superclass, it must be defined as public in the subclass.
  - if a method is defined as protected in the superclass, it can be re-defined as public in the subclass.

The `final` Modifier

- The `final` variable is a constant:

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

- The `final` class cannot be extended:

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

- The `final` method cannot be overridden by its subclasses.