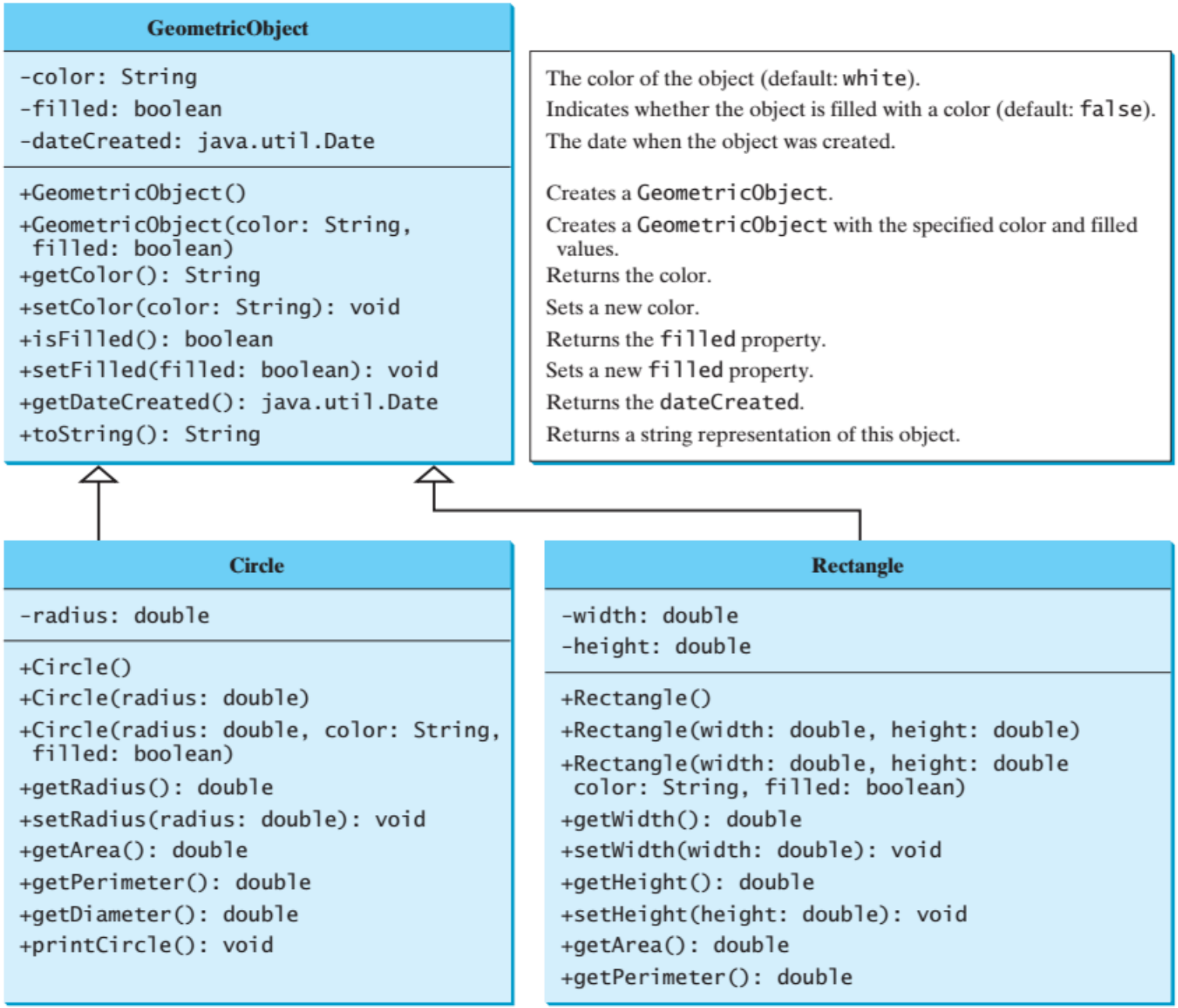
Inheritance and Polymorphism

Superclasses and Subclasses



Are superclass's Constructor Inherited?

* 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. *super* can also be used to call a superclass method.

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.

```
public ClassName() {
  // some statements
}
```

Equivalent

```
public ClassName() {
  super();
  // some statements
}
```

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

Equivalent

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

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

(1) Person's no-arg constructor is invoked

(2) Invoke Employee's overloaded constructor

(3) Employee's no-arg constructor is invoked

(4) Faculty'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");
  }
}
```

2. Invoke Faculty constructor

(1) Person's no-arg constructor is invoked
(2) Invoke Employee's overloaded constructor
(3) Employee's no-arg constructor is invoked
(4) Faculty'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");
  }
}
```

3. Invoke Employee's no-arg constructor

(1) Person's no-arg constructor is invoked

(2) Invoke Employee's overloaded constructor

(3) Employee's no-arg constructor is invoked

(4) Faculty'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");
  }
}
```

4. Invoke Employee(String) constructor

(1) Person's no-arg constructor is invoked
(2) Invoke Employee's overloaded constructor
(3) Employee's no-arg constructor is invoked
(4) Faculty'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) Person's no-arg constructor is invoked

(2) Invoke Employee's overloaded constructor

(3) Employee's no-arg constructor is invoked

(4) Faculty'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");
  }
}
```

(1) Person's no-arg constructor is invoked

(2) Invoke Employee's overloaded constructor

(3) Employee's no-arg constructor is invoked

(4) Faculty'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");
  }
}
```

(1) Person's no-arg constructor is invoked

(2) Invoke Employee's overloaded constructor

(3) Employee's no-arg constructor is invoked

(4) Faculty'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");
  }
}
```

(1) Person's no-arg constructor is invoked

(2) Invoke Employee's overloaded constructor

(3) Employee's no-arg constructor is invoked

(4) Faculty'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");
  }
}
```

(1) Person's no-arg constructor is invoked

(2) Invoke Employee's overloaded constructor

(3) Employee's no-arg constructor is invoked

(4) Faculty's no-arg constructor is invoked

9. Execute println

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

```
public void printCircle() {
  System.out.println("The circle is created " +
    super.getDateCreated() + " and the radius is " + radius);
}
```

Overriding Methods in the 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.

* 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.

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 ClassName {
  ...
}
```

Equivalent

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

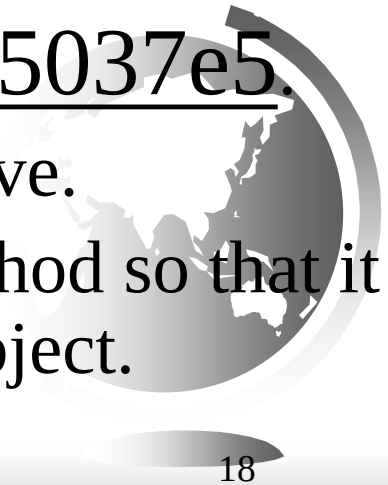
The toString() method in Object

* The toString() method returns a string representation of the object.

- The default implementation returns a string consisting of a class name, 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.

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

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

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.

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

* 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.
* These classes have their own implementation of the toString method.
* Which implementation is used will be determined dynamically by the JVM at runtime, called *dynamic binding*.

Dynamic Binding

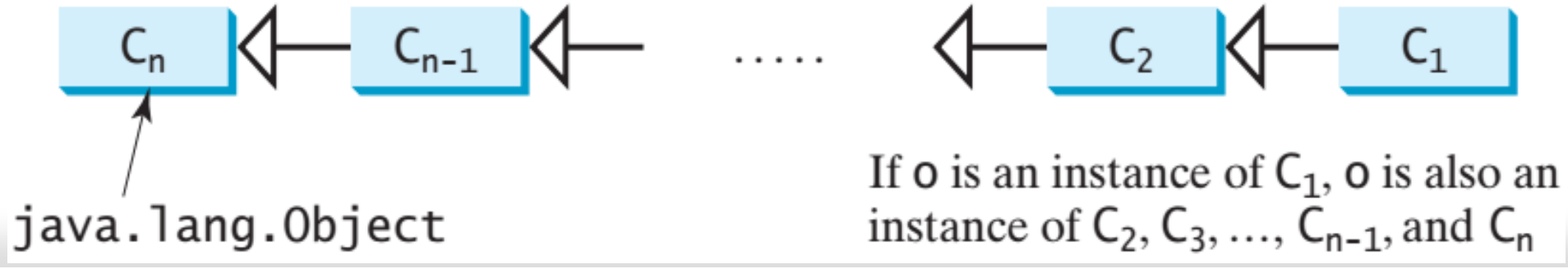
* Dynamic binding works as follows:

- 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.

- 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.



Casting Objects

* *Casting* can 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);
```

* However, if you want to assign the object o to the Student type using the following statement:

```
Student b = o;
```

- A compile error would occur.

Why Casting Is Necessary?

* Why

Object o = new Student() is ok, and

Student b = o fail ?

- 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 explicit casting.:

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

for this to be valid, o must be really a Student type, that is, o is created by: Object o = new Student();

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());
  ...
}
```

The `equals` Method

* The default implementation of the equals method in the Object class is as follows:

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

* The `equals()` method can be overridden to compare the contents of two objects.

```
public boolean equals(Circle c) {
    return radius == c.radius;
  }
  else
    return false;
}
```

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 store 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 first element o from this list. Returns true if an element is removed.
+size(): int	Returns the number of elements in this list.
+remove(index: int): boolean	Removes the element at the specified index. Returns true if an element is removed.
+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.

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

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

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))));
```

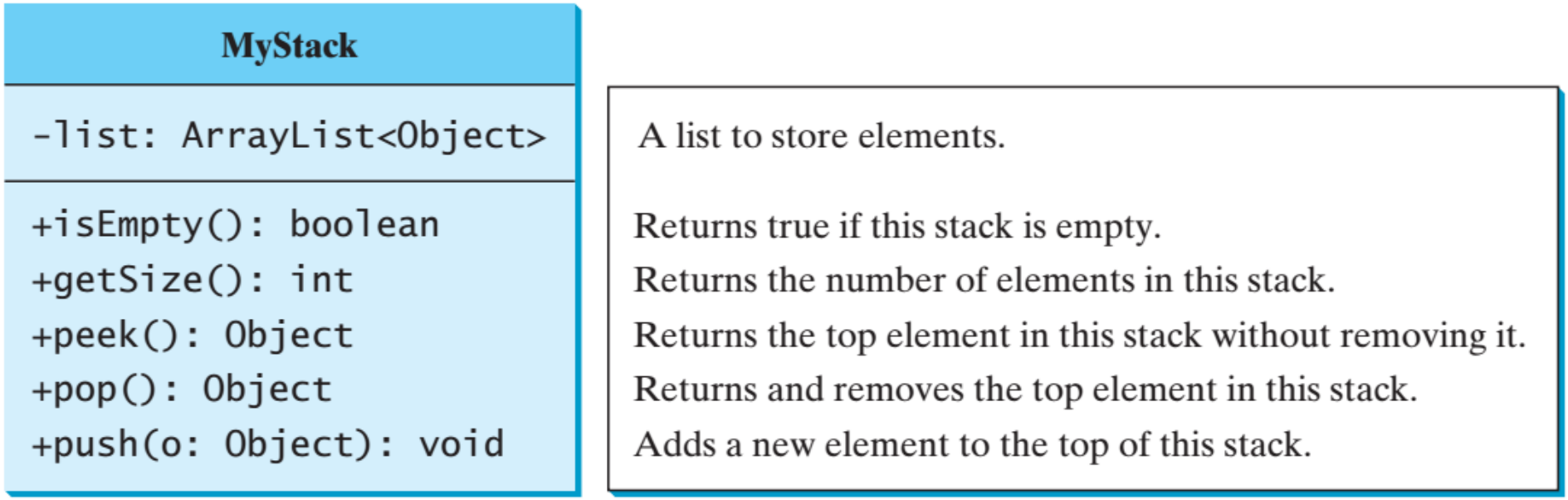
```
String[] array = {"red", "green", "blue"};

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);
```

The MyStack Classes

A stack to hold objects.



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

private, default (no modifier), protected, public

Accessibility Summary

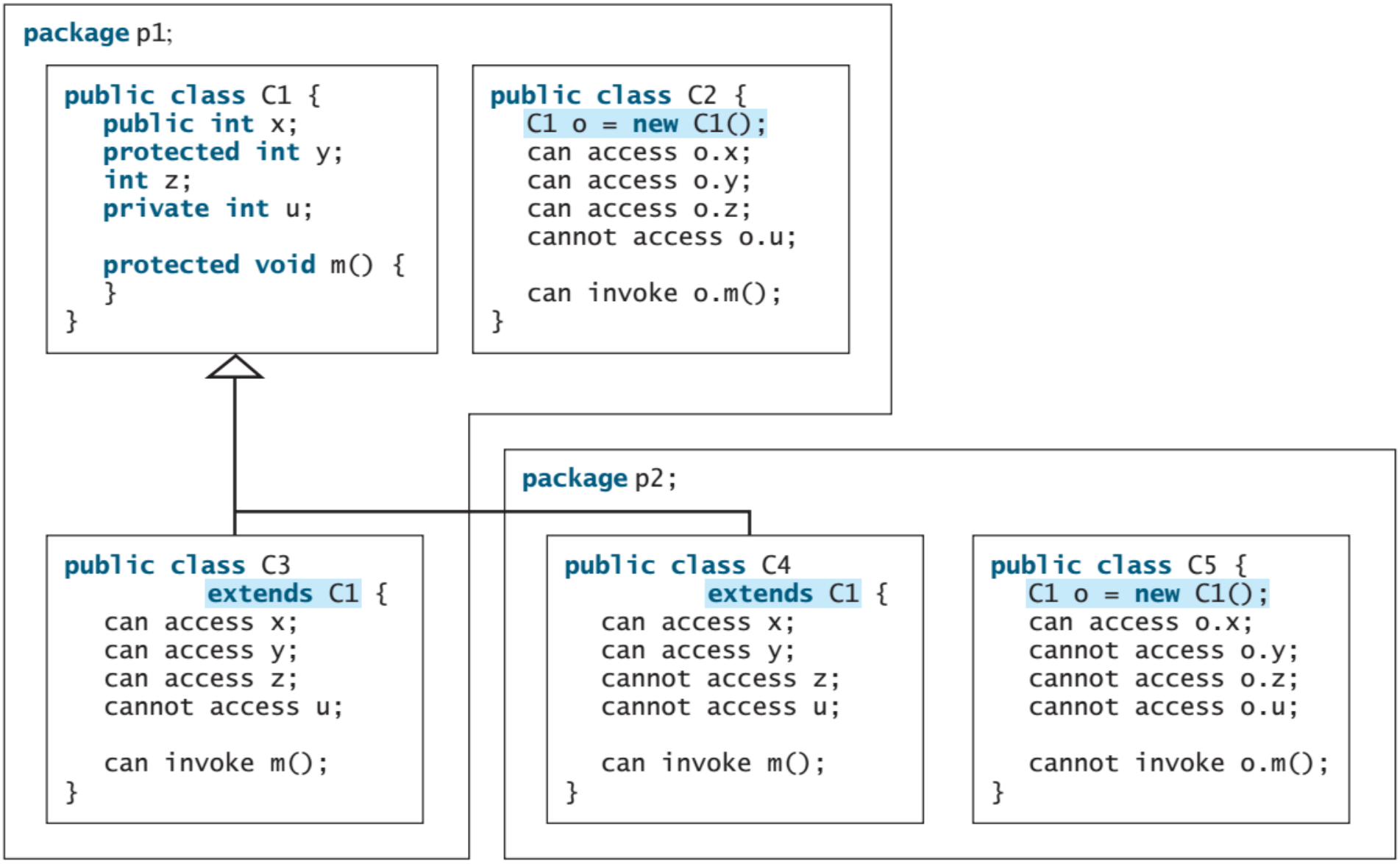
Modifier on members in a class	*Accessed from the same class*	*Accessed from the same package*	*Accessed from a subclass in a different package*	*Accessed from a different package*
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default (no modifier)	✓	✓	–	–
private	✓	–	–	–

A Subclass Cannot Weaken the Accessibility

* A subclass may override a protected method in its superclass and change its visibility to public.

* if a method is defined as public in the superclass, it must be defined as public in the subclass.

Visibility Modifiers



The `final` Modifier

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` 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.