جامعة بيرزيت

BIRZEIT UNIVERSITY

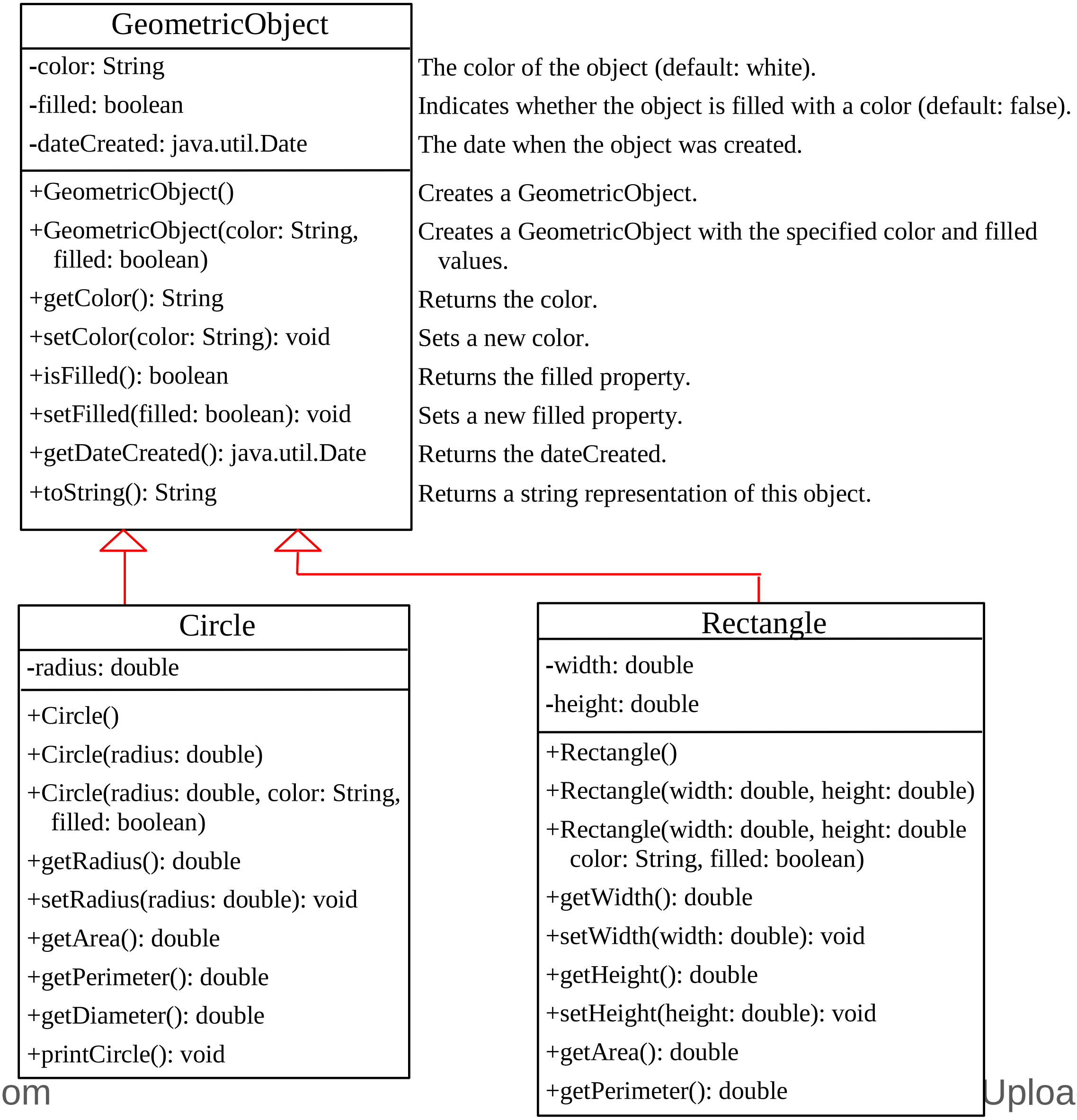
Inheritance and Polymorphism

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.

Superclasses and Subclasses



GeometricObject

-color: String — The color of the object (default: white).

-filled: boolean — Indicates whether the object is filled with a color (default: false).

-dateCreated: java.util.Date — The date when the object was created.

+GeometricObject() — Creates a GeometricObject.

+GeometricObject(color: String, filled: boolean) — Creates a GeometricObject with the specified color and filled values.

+getColor(): String — Returns the color.

+setColor(color: String): void — Sets a new color.

+isFilled(): boolean — Returns the filled property.

+setFilled(filled: boolean): void — Sets a new filled property.

+getDateCreated(): java.util.Date — Returns the dateCreated.

+toString(): String — Returns a string representation of this object.

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

Are Superclass’s Constructor Inherited?

❖ **No**. Unlike properties and methods, a superclass's **constructors are not inherited** in the subclass.

❖ They are invoked **explicitly** or **implicitly**.

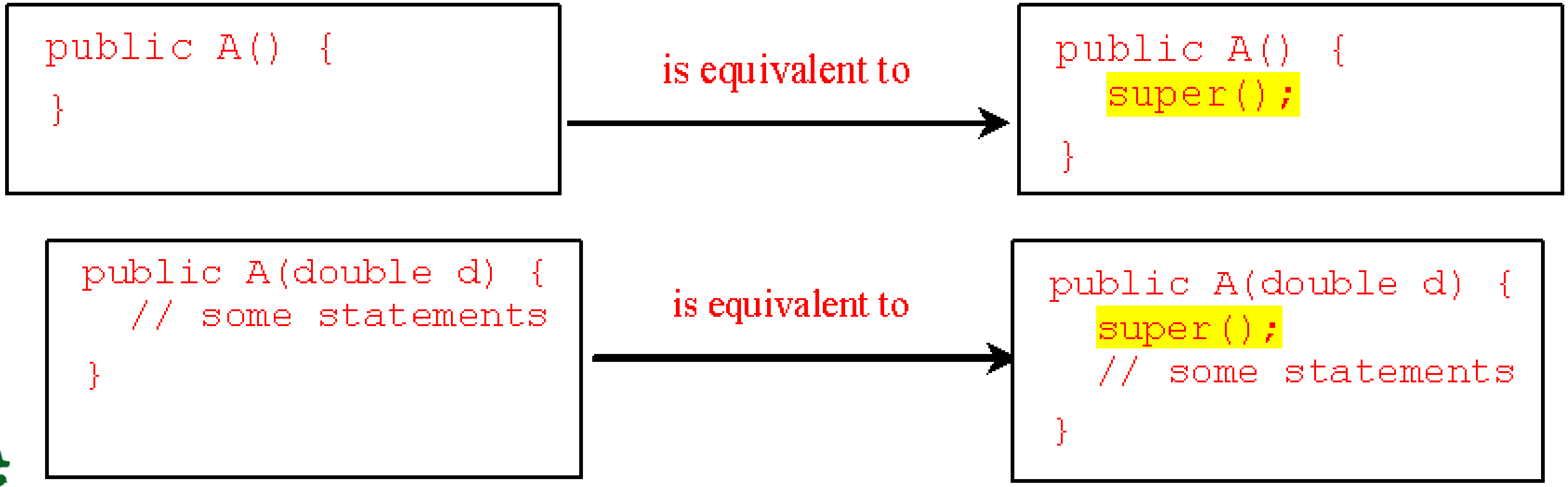
❖ Explicitly using the **super** keyword.

❖ 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.

- **super** 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 called ***constructor chaining***.

```
            public class Faculty extends Employee {
              public static void main(String[] args) {
                 Faculty f = new Faculty();
              }
              public Faculty() {
Super(); →      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) {
Super(); →      System.out.println(s);
              }
            }

            class Person {
              public Person() {
Super(); →      System.out.println("(1) Person's no-arg constructor is invoked");
              }
            }
```

Example on the Impact of a Superclass without no-arg Constructor

- Find out the **errors** in the following program:

```
public class Apple extends Fruit {
}

public 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.**

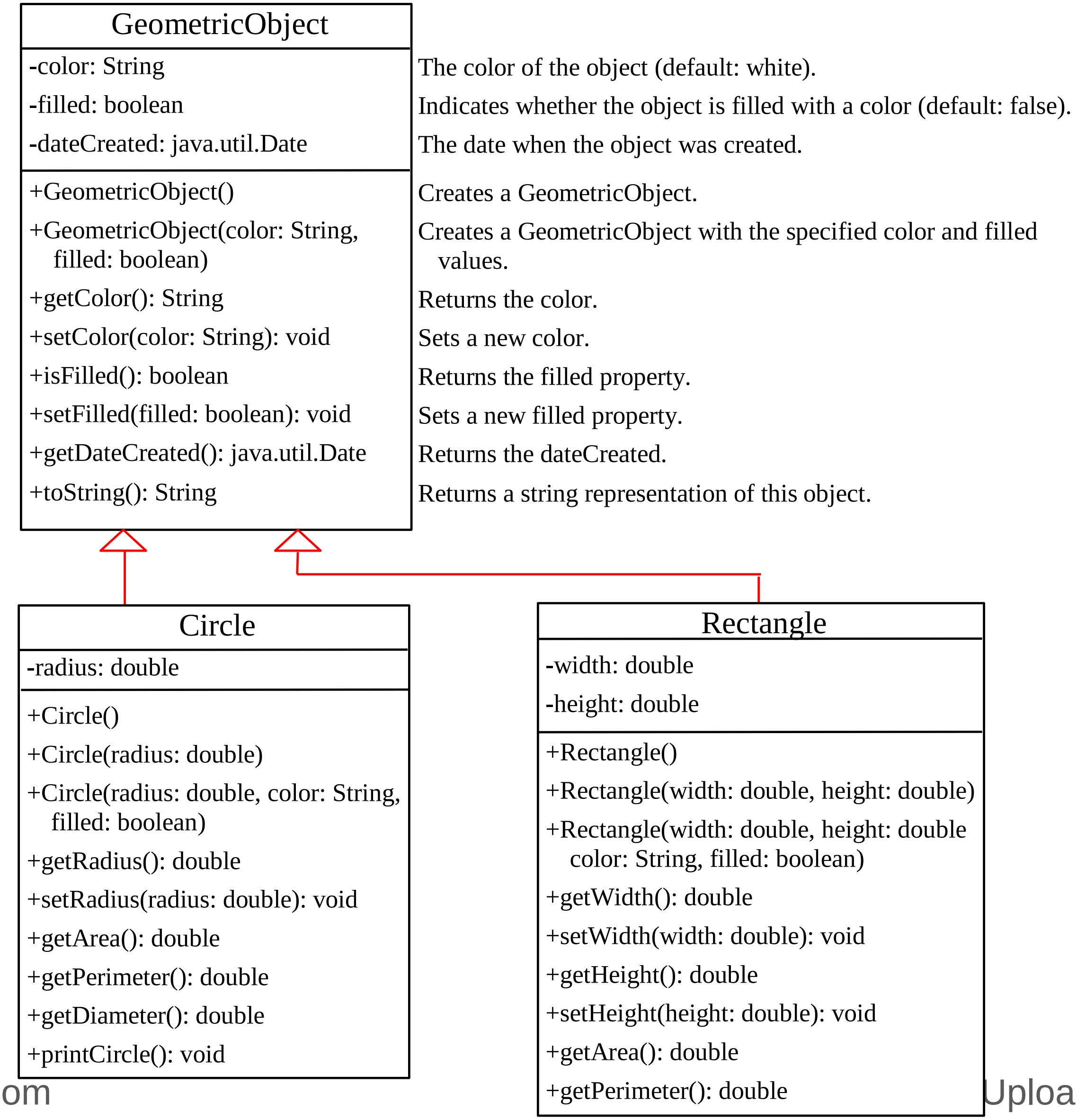
☞ **Override the methods of the superclass.**

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);
}
```

Superclasses and Subclasses



GeometricObject

-color: String — The color of the object (default: white).

-filled: boolean — Indicates whether the object is filled with a color (default: false).

-dateCreated: java.util.Date — The date when the object was created.

+GeometricObject() — Creates a GeometricObject.

+GeometricObject(color: String, filled: boolean) — Creates a GeometricObject with the specified color and filled values.

+getColor(): String — Returns the color.

+setColor(color: String): void — Sets a new color.

+isFilled(): boolean — Returns the filled property.

+setFilled(filled: boolean): void — Sets a new filled property.

+getDateCreated(): java.util.Date — Returns the dateCreated.

+toString(): String — Returns a string representation of this object.

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

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() + "\n radius 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 cont.

- 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 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);
  }
}
```

Overriding vs. Overloading

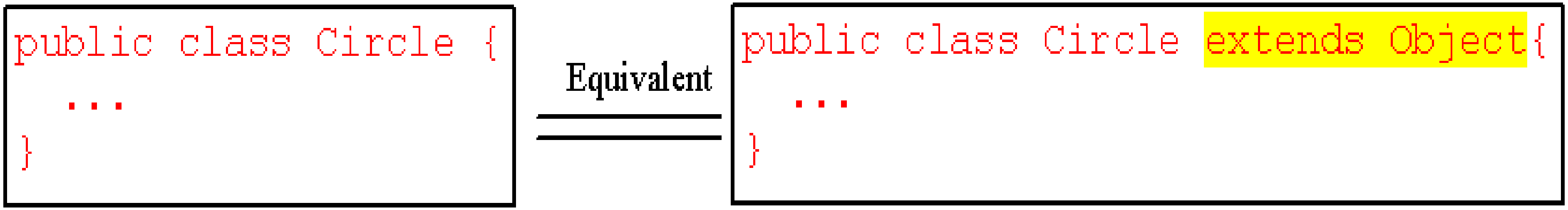
```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

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

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

The **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**.



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

```
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 (**@**).
  - A **number** representing this object.

The toString() method in Object

```
Circle c = new Circle();

System.out.println(c.toString());
```

- The code displays something like:

```
Circle@15037e5
```

- This message is not very helpful or informative.
- Usually you should **override** the **toString** method so that it returns an informative string representing the object.

```
class GraduateStudent extends Student {
}

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

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

Polymorphism

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

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

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**.

Dynamic Binding

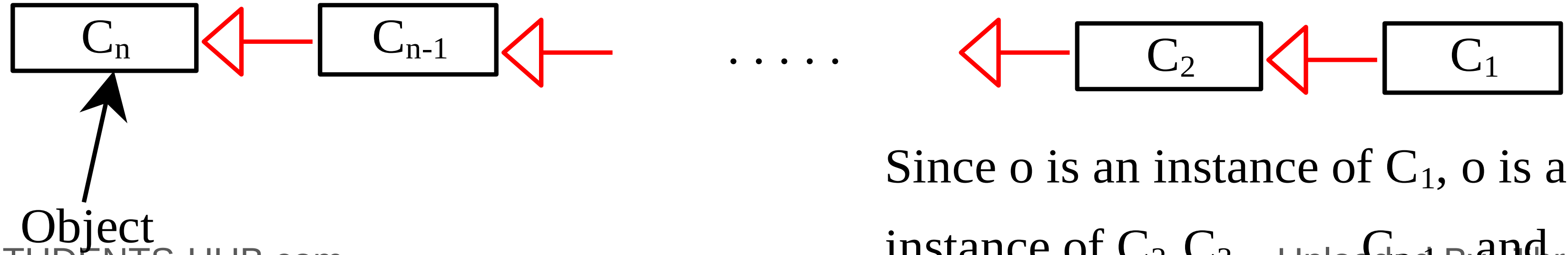
```
public class Demo {
  public static void main(String[] a) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }
  public static void m(Object x) {
       System.out.println(x.toString());
  }
}
```

This capability is known as ***dynamic binding***.

- 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 JVM at **runtime**.

Dynamic Binding

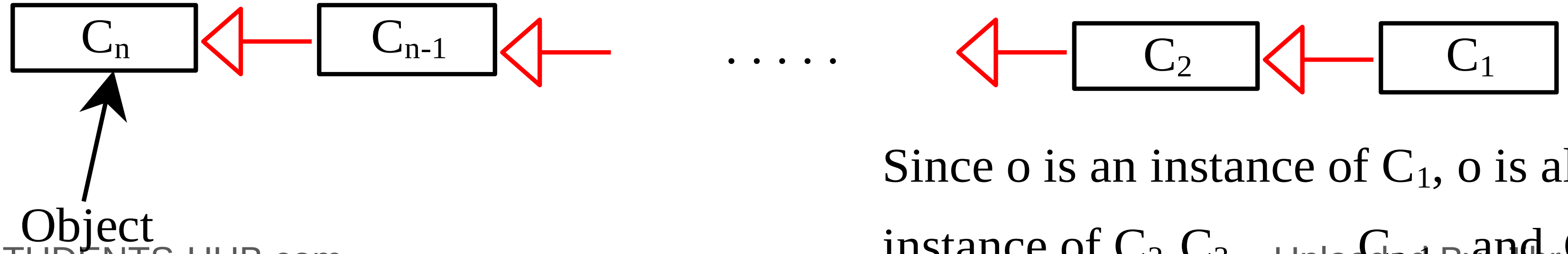
- Dynamic binding works as follows:
  - Suppose an object **o** is an instance of classes $\mathbf{C_1}$, $\mathbf{C_2}$, ..., $\mathbf{C_{n-1}}$, and $\mathbf{C_n}$, where $\mathbf{C_1}$ is a subclass of $\mathbf{C_2}$, $\mathbf{C_2}$ is a subclass of $\mathbf{C_3}$, ..., and $\mathbf{C_{n-1}}$ is a subclass of $\mathbf{C_n}$.
  - That is, $\mathbf{C_n}$ is the most general class, and $\mathbf{C_1}$ 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

Dynamic Binding cont.

- Dynamic binding works as follows:
  - If **o** invokes a method **p**, the **JVM** searches the implementation for the method **p** in $\mathbf{C_1}$, $\mathbf{C_2}$, **...,** $\mathbf{C_{n-1}}$ and $\mathbf{C_n}$, 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

Generic Programming

```
public class Demo {
  public static void main(String[] a) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }
  public static void m(Object x){
      System.out.println(x.toString());
  }
}
```

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**).
- ❖ When an **object** (e.g., a **Student** 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

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

```
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.

Why Casting Is Necessary?

- 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:

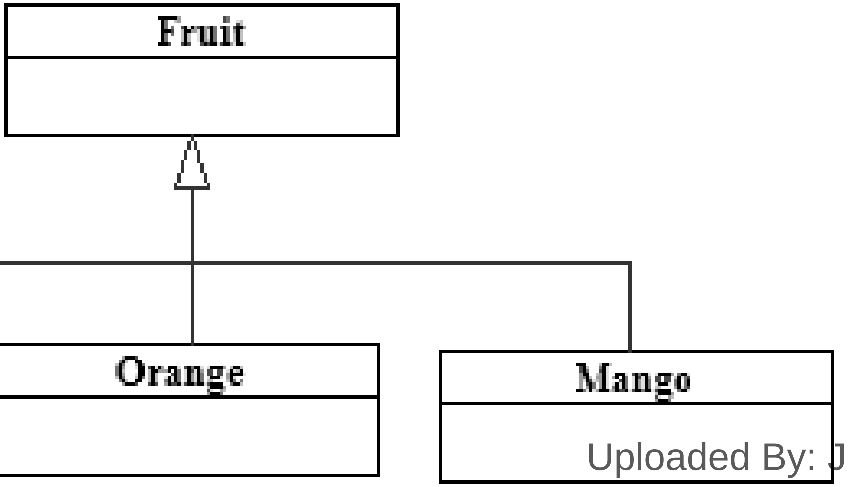
Student b = (Student) o ; // **Explicit casting**

Casting from Superclass to Subclass

- Explicit casting **must** be used when casting an object from a superclass to a subclass.

```
Fruit fruit = new Apple();
Apple a = (Apple) fruit;
Orange o = (Orange) fruit;
```

- This type of casting **may not** always succeed.



The instanceof Operator

❖ Use 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()   );
}
```

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 );
}
```

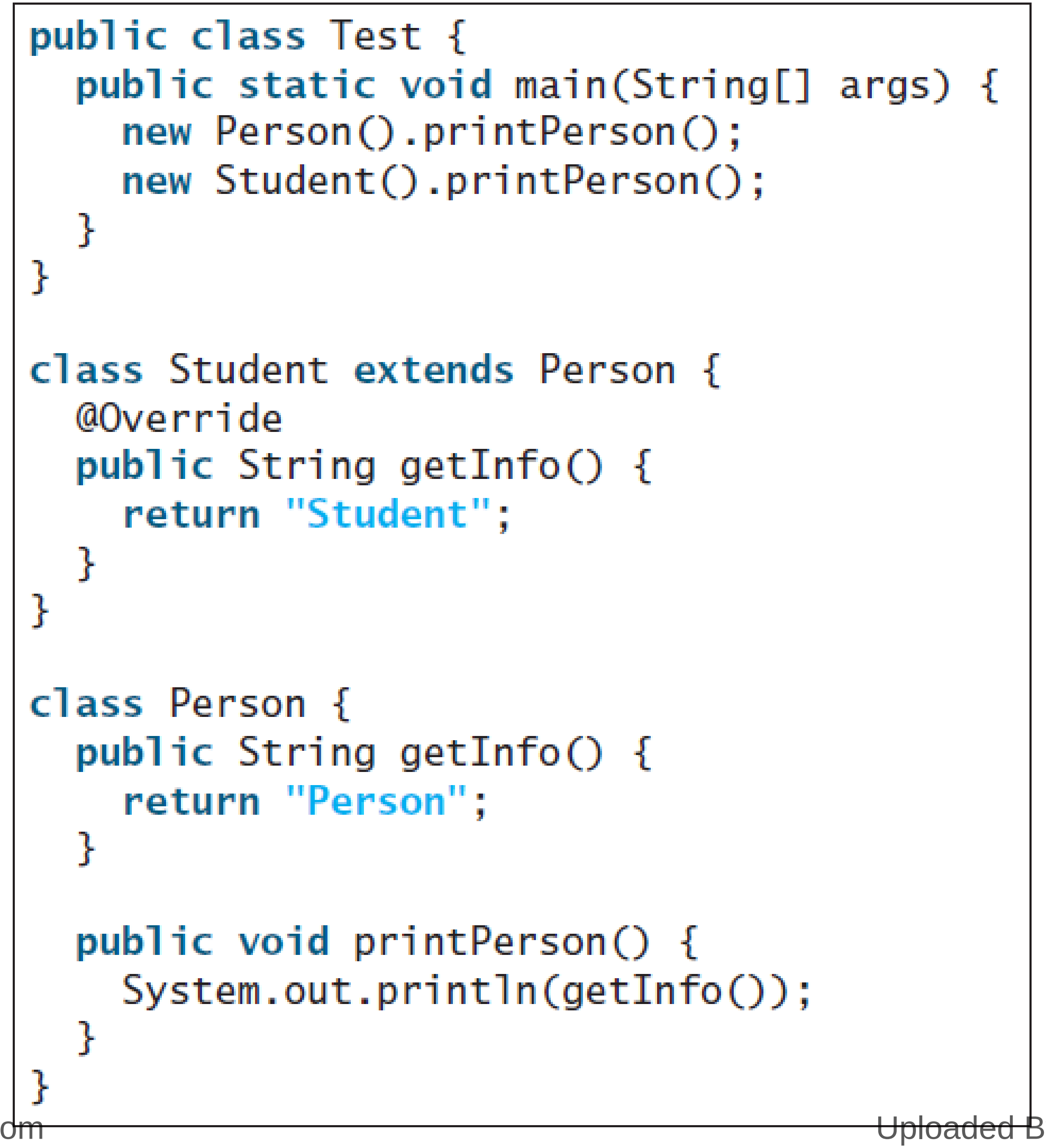
- For example, the **equals** method is **overridden** in the **Circle** class.

```
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.

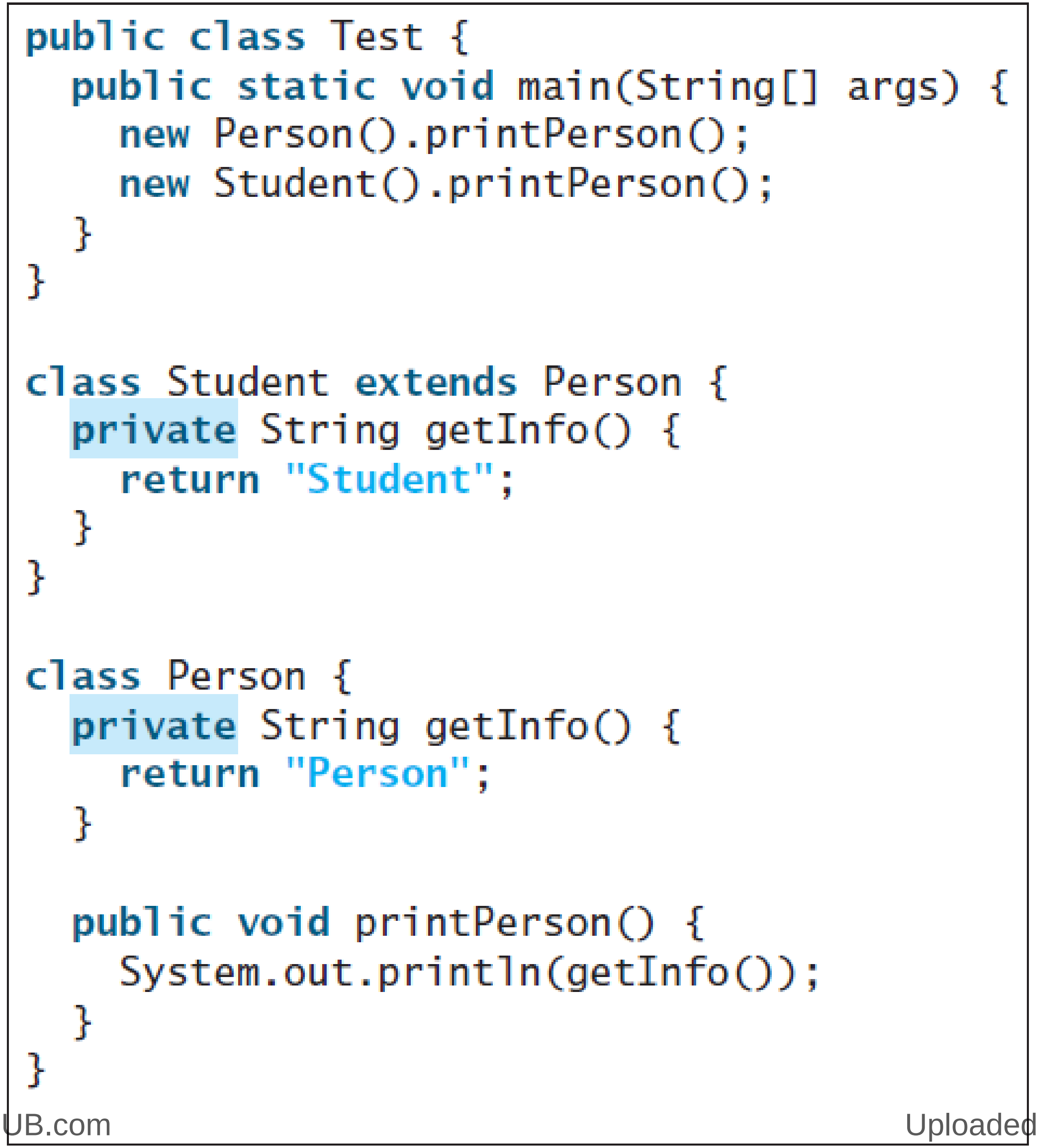


```java
public class Test {
  public static void main(String[] args) {
    new Person().printPerson();
    new Student().printPerson();
  }
}

class Student extends Person {
  @Override
  public String getInfo() {
    return "Student";
  }
}

class Person {
  public String getInfo() {
    return "Person";
  }

  public void printPerson() {
    System.out.println(getInfo());
  }
}
```



```java
public class Test {
  public static void main(String[] args) {
    new Person().printPerson();
    new Student().printPerson();
  }
}

class Student extends Person {
  private String getInfo() {
    return "Student";
  }
}

class Person {
  private String getInfo() {
    return "Person";
  }

  public void printPerson() {
    System.out.println(getInfo());
  }
}
```

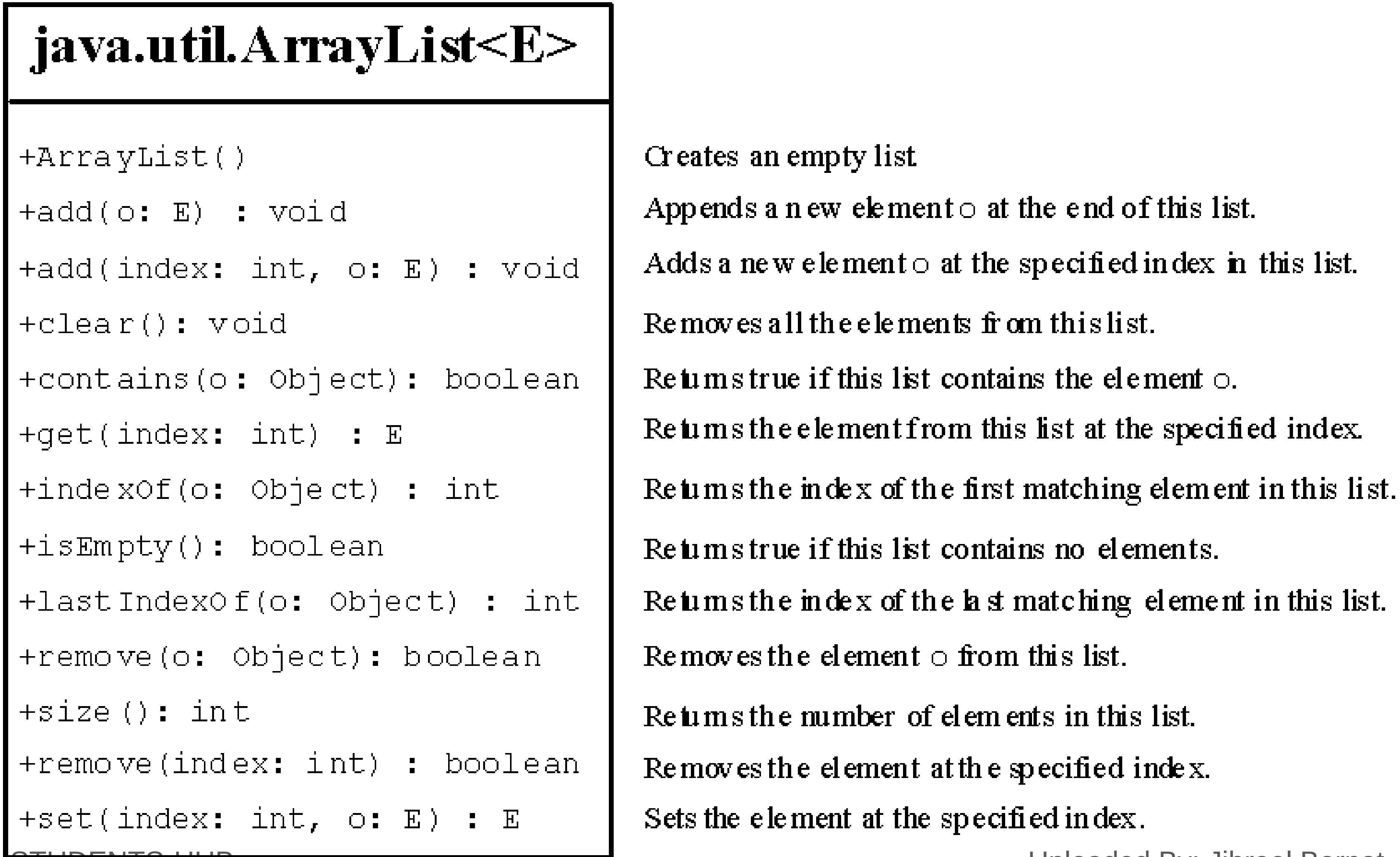
The ArrayList Class

- 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.

The ArrayList Class



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 <E>

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

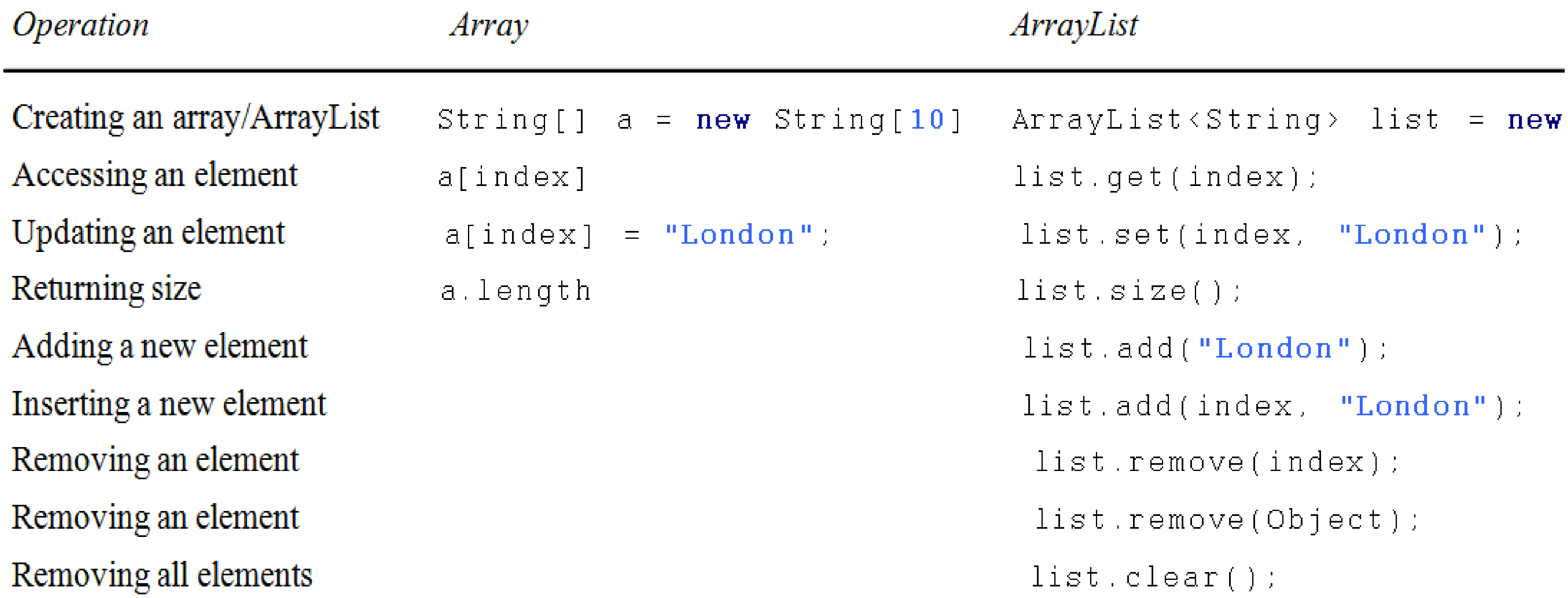
- You can specify a concrete type to replace **E** when creating an **ArrayList**.

- For example, the following statement creates an **ArrayList** and assigns its reference to variable **cities**. This **ArrayList** object can be used to store **strings:**

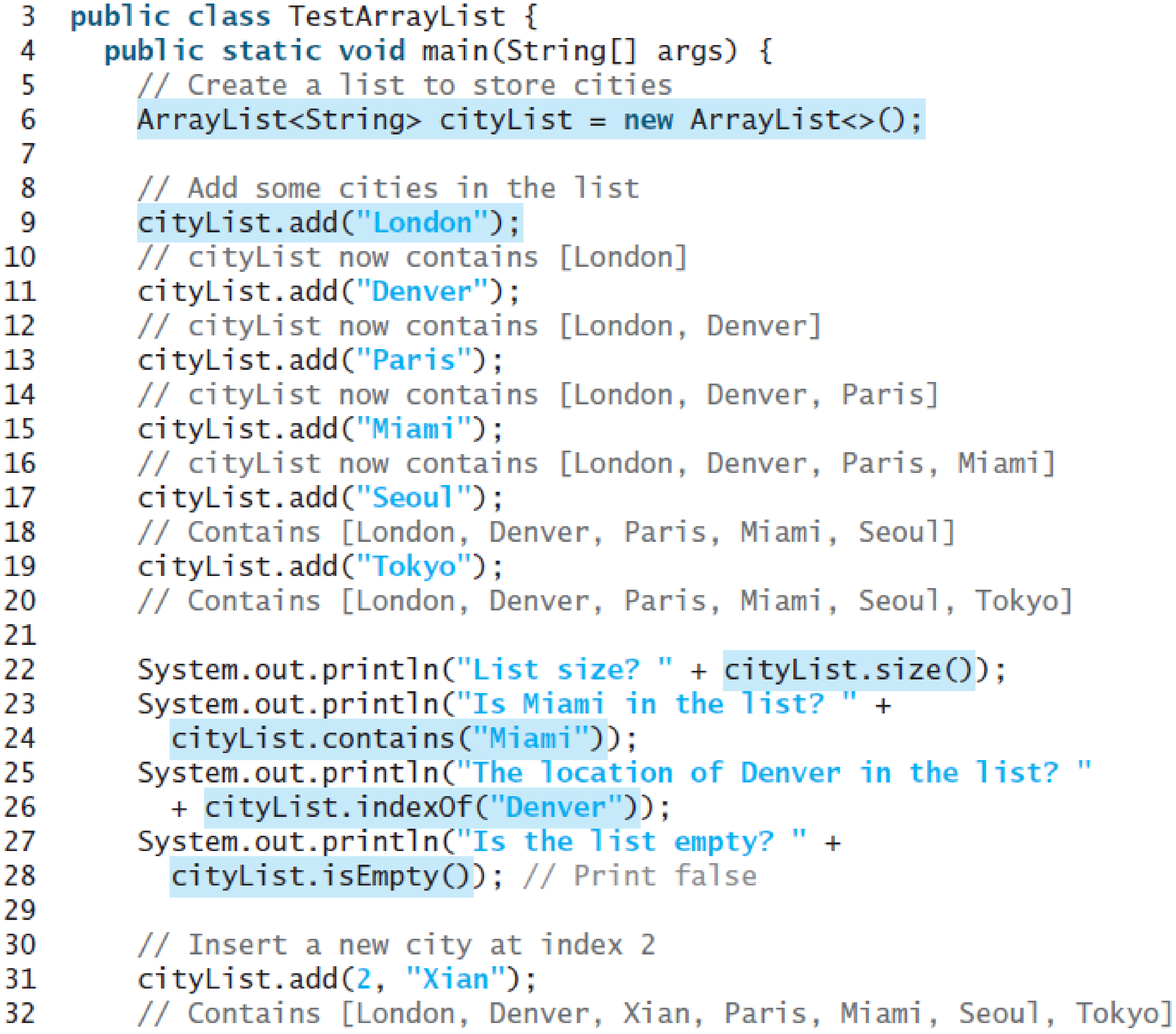
```
ArrayList<String>  cities  =  new ArrayList<String>();

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

Differences and Similarities between Arrays and ArrayList



Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new`
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();`

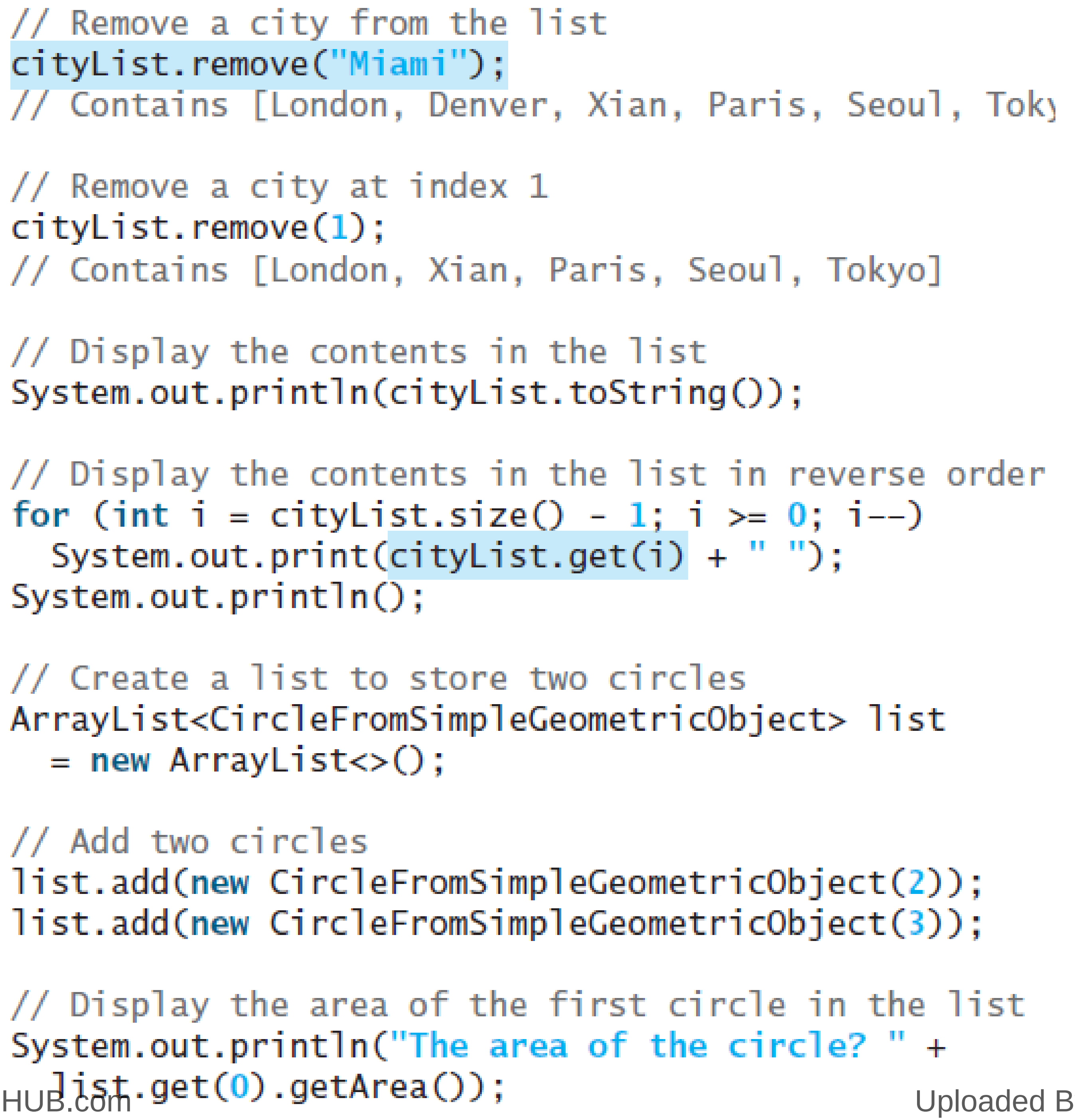


```
public class TestArrayList {
  public static void main(String[] args) {
    // Create a list to store cities
    ArrayList<String> cityList = new ArrayList<>();

    // 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, Toky

// Remove a city at index 1
cityList.remove(1);
// Contains [London, Xian, Paris, Seoul, Tokyo]

// Display the contents in the list
System.out.println(cityList.toString());

// Display the contents in the list in reverse order
for (int i = cityList.size() - 1; i >= 0; i--)
  System.out.print(cityList.get(i) + " ");
System.out.println();

// Create a list to store two circles
ArrayList<CircleFromSimpleGeometricObject> list
  = new ArrayList<>();

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

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

ArrayLists 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 ArrayList

```
java.util.Collections.max(list)

java.util.Collections.min(list)
```

Shuffling an ArrayList

```
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 **protected** Modifier

- The **protected** modifier can be applied on **data** and **methods** in a class.
- A **protected** data/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.

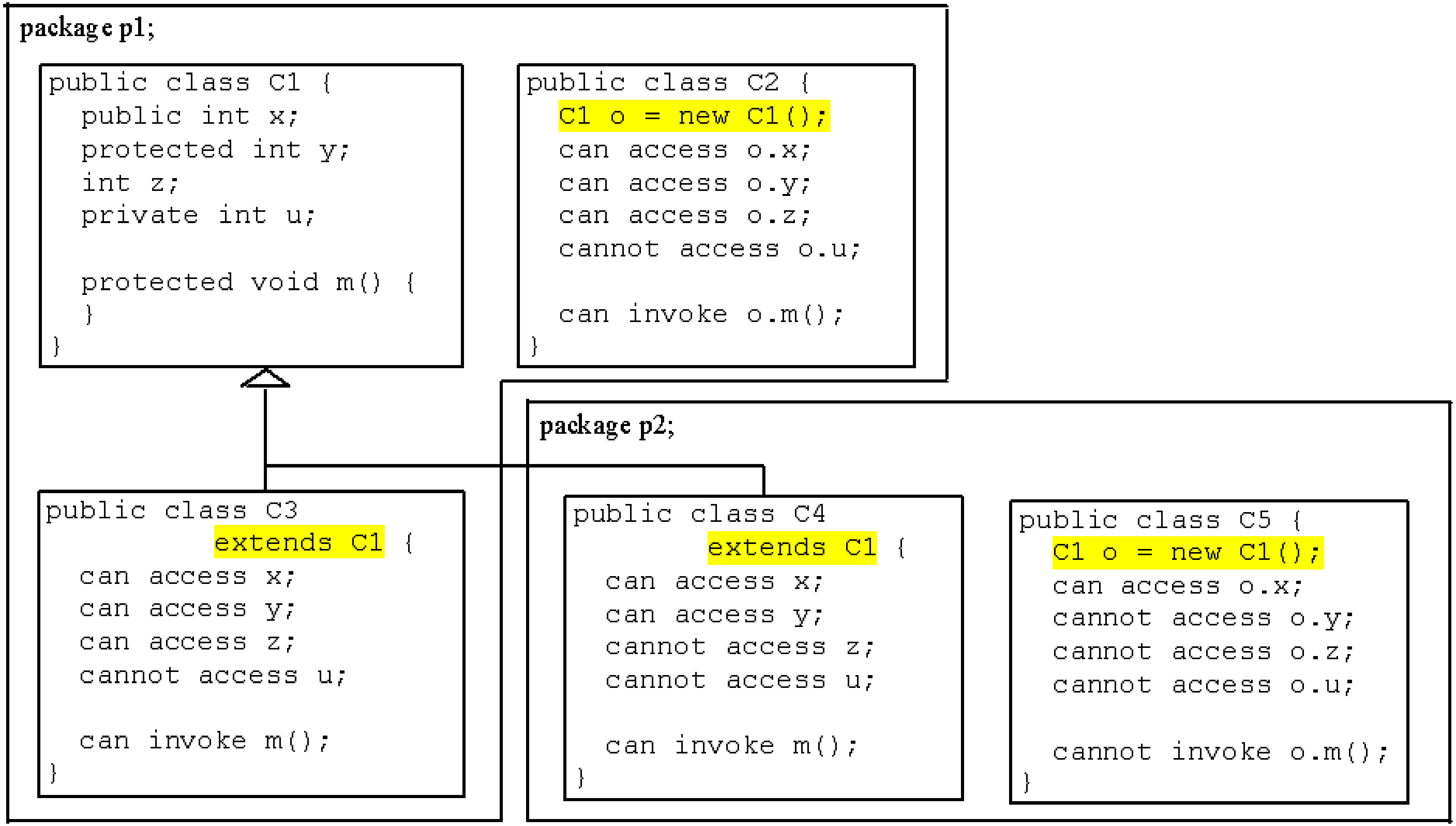
Visibility increases →

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.

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.

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.