COMP30820 Java Programming (Conv)

Michael O’Mahony

Chapter 11 Inheritance and Polymorphism

Objectives

- To define a subclass from a superclass through inheritance (§11.2).
- To invoke the superclass's constructors and methods using the `super` keyword (§11.3).
- To override instance methods in the subclass (§11.4).
- To distinguish differences between overriding and overloading (§11.5).
- To introduce the `Object` class and its `toString()` method (§11.6).
- To discover polymorphism and dynamic binding (§§11.7–11.8).
- To describe casting and explain why explicit is necessary (§11.9).
- To explore the `equals` method in the `Object` class (§11.10).
- To enable data and methods in a superclass to be accessible from subclasses using the `protected` visibility modifier (§11.14).
- To prevent class extending and method overriding using the `final` modifier (§11.15).

Inheritance

Suppose you wish 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

Inheritance is an important and powerful feature for reusing software.

Object-oriented programming (OOP) allows you to define new classes from existing classes – this is called *inheritance*.

Inheritance is used to model *is-a* relationships.

Inheritance enables you to define a *general* class (i.e. a *superclass*) and later *extend* it to more *specialized* classes (i.e. *subclasses*).

The specialized classes inherit the properties and methods from the general class.

Inheritance

Example − designing classes to model geometric objects such as circles and rectangles.

Geometric objects have many common properties and behaviours. For example, they may have a color and can be filled or unfilled:

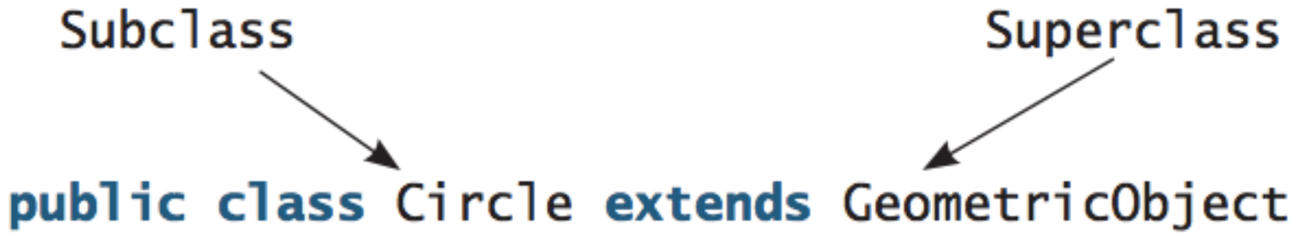
- A general class `GeometricObject` can be used to model all geometric objects.
- This class contains the properties `color` and `filled` and their appropriate getter and setter methods.

Circles and rectangles are special types of geometric object – they share common properties and methods with other geometric objects:

- Thus the `Circle` and `Rectangle` classes can be defined as subclasses of the `GeometricObject` class.

Inheritance – Syntax

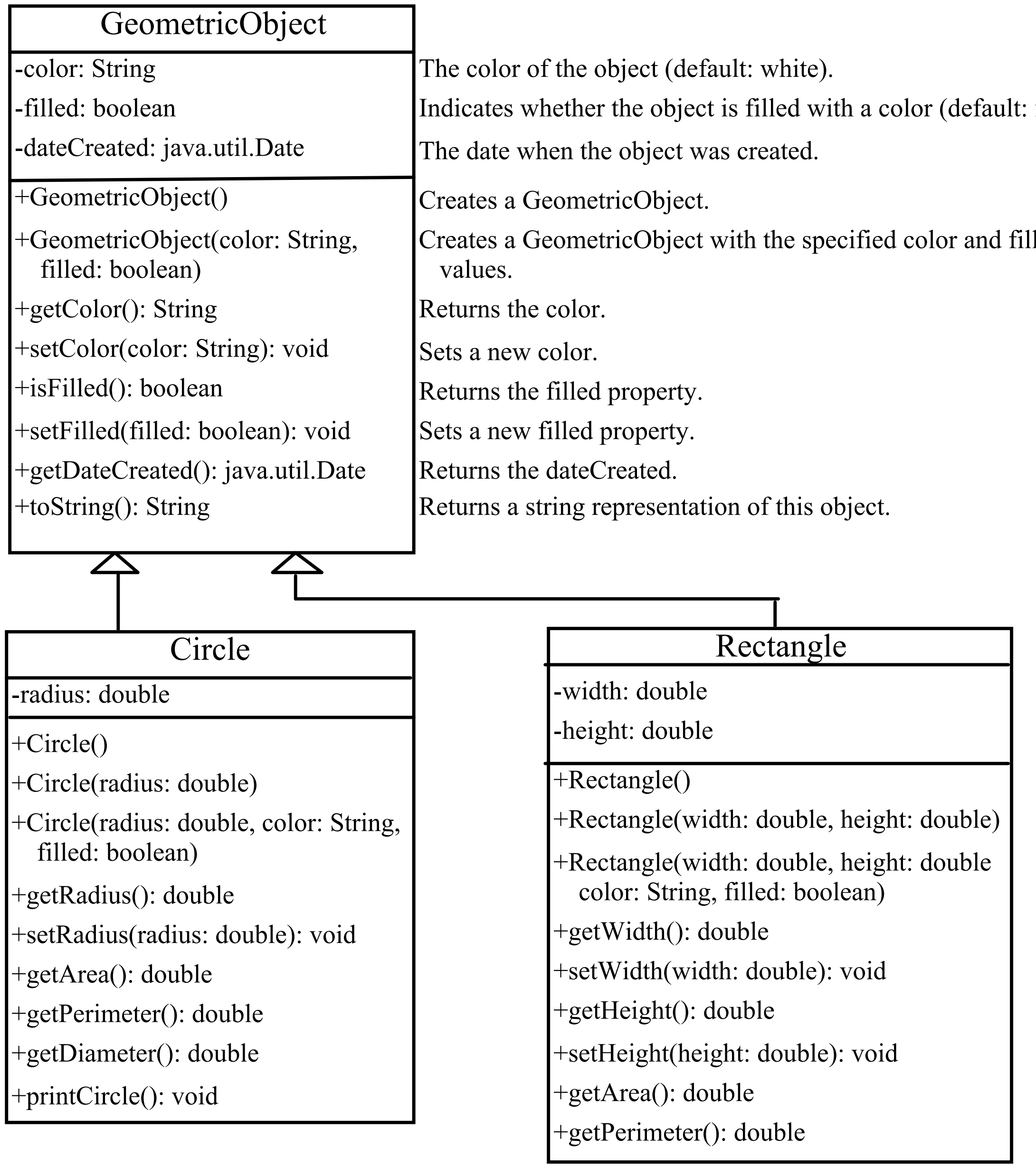
The `Circle` class extends the `GeometricObject` class using the following syntax:



```
public class Circle extends GeometricObject
```

The keyword `extends` tells the compiler that the `Circle` class is a subclass of the `GeometricObject` class.

Superclasses and Subclasses – Example


GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
+GeometricObject()
+GeometricObject(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
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

The color of the object (default: white).
Indicates whether the object is filled with a color (default: false).
The date when the object was created.
Creates a GeometricObject.
Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns the dateCreated.
Returns a string representation of this object.

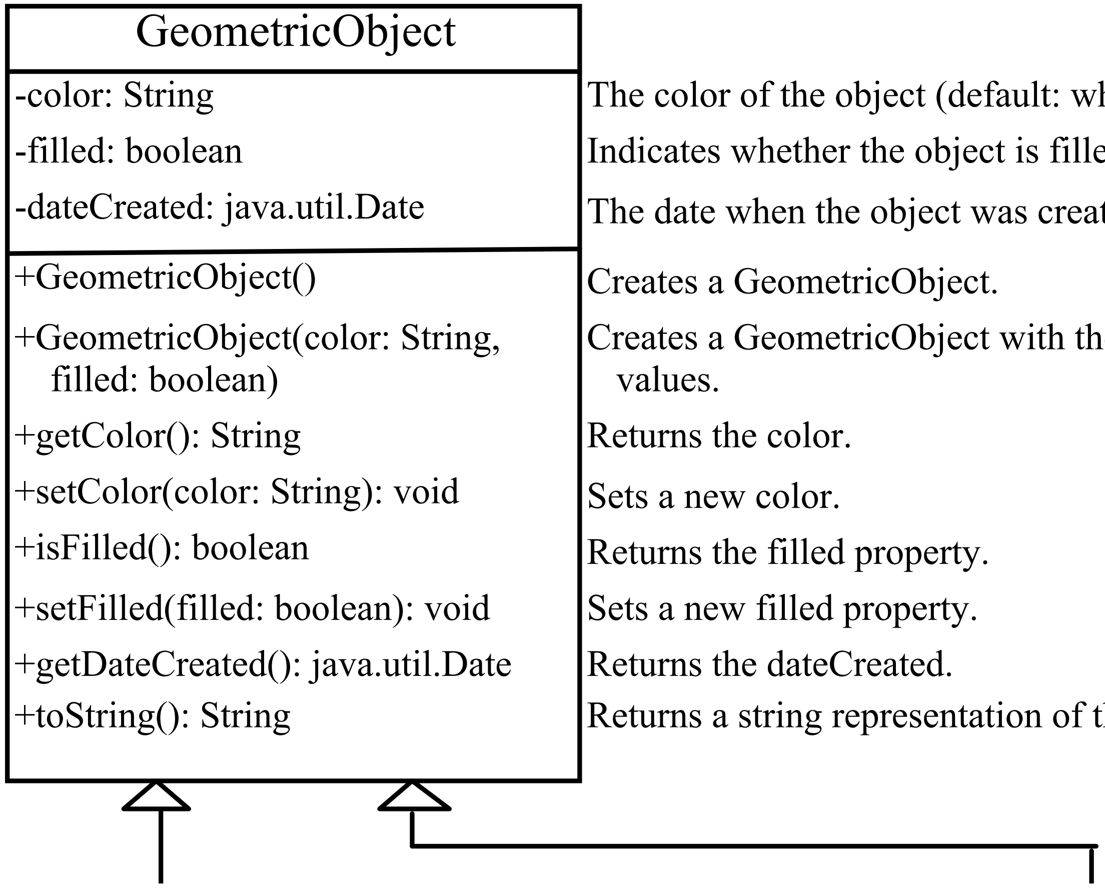
GeometricObject

Circle

Rectangle

TestCircleRectangle

Superclasses and Subclasses – Example


GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
+GeometricObject()
+GeometricObject(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String

The color of the object (default: white).

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

The date when the object was created.

Creates a GeometricObject.

Creates a GeometricObject with the specified color and filled values.

Returns the color.

Sets a new color.

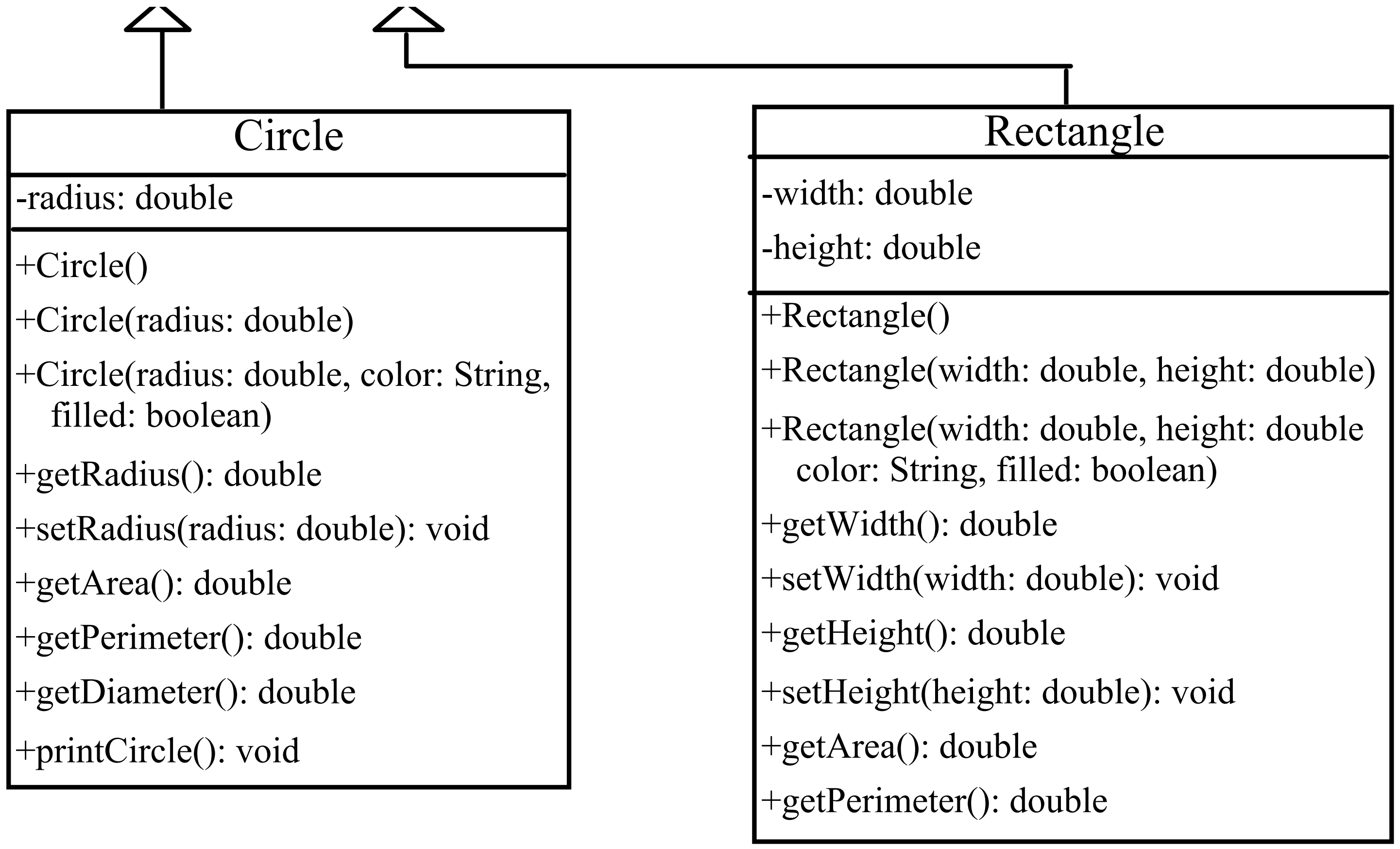
Returns the filled property.

Sets a new filled property.

Returns the dateCreated.

Returns a string representation of this object.

Superclasses and Subclasses – Example



Superclasses and Subclasses – Example

The `Circle` class inherits all accessible data fields and methods from the `GeometricObject` class:

- The `Circle` class inherits the methods `getColor`, `setColor`, `isFilled`, `setFilled`, `getDateCreated`, and `toString`.
- In addition, it has a new data field `radius` and associated getter/setter methods. It also contains the `getArea()`, `getPerimeter()`, and `getDiameter()` methods for returning the area, perimeter, and diameter of the circle.

Likewise, the `Rectangle` class inherits all accessible data fields and methods from the `GeometricObject` class:

- The `Rectangle` class inherits the methods `getColor`, `setColor`, `isFilled`, `setFilled`, `getDateCreated`, and `toString`.
- In addition, it has the new data fields `width` and `height` and associated getter and setter methods. It also contains the `getArea()` and `getPerimeter()` methods for returning the area and perimeter of the rectangle.

Superclasses and Subclasses – Example

Private data fields in a superclass are not accessible outside the class.

Therefore, they cannot be used directly in a subclass.

They can, however, be accessed/mutated through public getter/setter methods (provided that such methods are defined in the superclass).

Superclasses and Subclasses – Example

Private data fields in a superclass are not accessible outside the class.

Therefore, they cannot be used directly in a subclass.

They can, however, be accessed/mutated through public getter/setter methods (provided that such methods are defined in the superclass).

Example:

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

Superclasses and Subclasses – Example

Private data fields in a superclass are not accessible outside the class.

Therefore, they cannot be used directly in a subclass.

They can, however, be accessed/mutated through public getter/setter methods (provided that such methods are defined in the superclass).

Example:

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

The `super` Keyword

The keyword `super` refers to the superclass of the class in which `super` appears.

The `super` keyword can be used in two ways:

- To call a superclass constructor.
- To call a superclass method.

The `super` Keyword – Constructors

Unlike properties and methods, the constructors of a superclass are not inherited by a subclass.

Instead, they are invoked explicitly or implicitly.

The syntax to explicitly invoke a superclass's constructor is:

- `super()` – invokes the no-arg superclass constructor.
- `super(arguments)` – invokes the superclass constructor that matches the arguments.

The statement `super()` or `super(arguments)` must be the first statement of the subclass's constructor:

- This is the only way to explicitly invoke a superclass constructor.

The `super` Keyword – Constructors

For example, the following are equivalent:

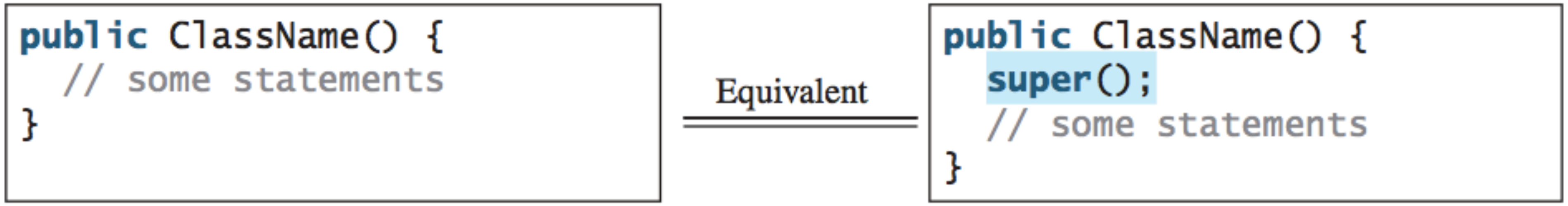
```
public Circle(double radius, String color, boolean filled) {
    this.radius = radius;
    setColor(color);
    setFilled(filled);
}
```

```
public Circle(double radius, String color, boolean filled) {
    super(color, filled); // must be the first statement
    this.radius = radius;
}
```

Constructor Chaining

A constructor in a class may invoke an overloaded constructor in the same class (using the keyword `this`) *or* its superclass constructor.

If neither is invoked explicitly, the compiler automatically puts `super()` as the first statement in the constructor. For example:



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

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

Constructor Chaining

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

When constructing an object of a subclass, the subclass constructor *first* invokes its superclass constructor (either explicitly or implicitly) before performing its own tasks.

If the superclass is derived from another class, the superclass constructor invokes its parent-class constructor before performing its own tasks.

This process continues until the last constructor along the inheritance hierarchy is called. This is called *constructor chaining*.

Example...

```
public class A {
   public A () {
      System.out.println("A");
   }
}
```

```
public class A {
   public A () {
      System.out.println("A");
   }
}
```

```
public class B extends A {
   public B () {
      System.out.println("B");
   }
}
```

```
public class A {
   public A () {
      System.out.println("A");
   }
}
```

```
public class B extends A {
   public B () {
      System.out.println("B");
   }
}
```

```
public class C extends B {
   public C () {
      System.out.println("C");
   }
}
```

```
public class A {
   public A () {
      System.out.println("A");
   }
}
```

```
public class B extends A {
   public B () {
      System.out.println("B");
   }
}
```

super();

```
public class C extends B {
   public C () {
      System.out.println("C");
   }
}
```

super();

```
public class A {
   public A () {
      System.out.println("A");
   }
}
```

```
public class B extends A {
   public B () {
      System.out.println("B");
   }
}
```

super();

```
public class C extends B {
   public C () {
      System.out.println("C");
   }
}
```

super();

```
public class Test {
   public static void main(String[] args) {
      C c = new C();
   }
}
```

```
public class A {
   public A () {
      System.out.println("A");
   }
}
```

```
public class B extends A {
   public B () {
      System.out.println("B");
   }
}
```

super();

```
public class C extends B {
   public C () {
      System.out.println("C");
   }
}
```

super();

```
public class Test {
   public static void main(String[] args) {
      C c = new C();
   }
}
```

Output:

```
A
B
C
```

Calling Superclass Methods

The keyword `super` is also used to reference a method in the superclass.

The general syntax is: `super.method(parameters);`

You could rewrite the `printCircle()` method in `Circle` class as follows:

Calling Superclass Methods

The keyword `super` is also used to reference a method in the superclass.

The general syntax is: `super.method(parameters);`

You could rewrite the `printCircle()` method in `Circle` class as follows:

```
public void printCircle() {
  System.out.println("created: " + getDateCreated() +
  "\n color: " + getColor() +
  "\n filled: " + isFilled() +
  "\n radius: " + radius);
}
```

Calling Superclass Methods

The keyword `super` is also used to reference a method in the superclass.

The general syntax is: `super.method(parameters);`

You could rewrite the `printCircle()` method in `Circle` class as follows:

```
public void printCircle() {
  System.out.println("created: " + getDateCreated() +
  "\n color: " + getColor() +
  "\n filled: " + isFilled() +
  "\n radius: " + radius);
}
```

```
public void printCircle() {
  System.out.println("created: " + super.getDateCreated() +
  "\n color: " + super.getColor() +
  "\n filled: " + super.isFilled() +
  "\n radius: " + radius);
}
```

Calling Superclass Methods

The keyword `super` is also used to reference a method in the superclass.

The general syntax is: `super.method(parameters);`

You could rewrite the `printCircle()` method in `Circle` class as follows:

```
public void printCircle() {
  System.out.println("created: " + getDateCreated() +
  "\n color: " + getColor() +
  "\n filled: " + isFilled() +
  "\n radius: " + radius);
}
```

```
public void printCircle() {
  System.out.println("created: " + super.getDateCreated() +
  "\n color: " + super.getColor() +
  "\n filled: " + super.isFilled() +
  "\n radius: " + radius);
}
```

It is not necessary use `super` in this case – because the methods are defined in class `GeometricObject` and are inherited by the `Circle` class. However there are some cases where use of the keyword `super` is needed.

Overriding Methods Defined in the Superclass

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*.

```
public class Circle extends GeometricObject {
  // Other methods are omitted

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

- To override a method, the method must be defined in the subclass using the *same signature and the same return type* as in its superclass.
- The override annotation (`@Override`) denotes that the annotated method is required to override a method in the superclass. If a method with this annotation does not override its superclass's method, the compiler will report an error.

Overriding Methods in the Superclass

The `toString` method is defined in the `GeometricObject` class and overridden in the `Circle` class:

- However, both `toString` methods can be used in the `Circle` class.
- To invoke the `toString` method defined in the `GeometricObject` class from the `Circle` class, use `super.toString().`
  - Second use of `super` keyword – using `super` to call a superclass method.

An instance method can be overridden only if it is accessible:

- 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.

Overriding Methods in the Superclass

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.
- Hidden static methods can be invoked using the syntax:

```
SuperClassName.staticMethodName(...)
```

Overriding vs. Overloading

Overloading:

- Means to define multiple methods with the same name but different signatures, for example:
  - `int max(int x, int y)`
  - `double max(double x, double y)`
- Overloaded methods can be either in the same class or in different classes related by inheritance.

Overriding:

- Means to provide a new implementation for a method in a subclass.
- Overridden methods are in different classes related by inheritance.

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`.

For example, the following two class definitions are the same:

```
public class Loan {
  ...
}
```

Equivalent

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

For example, the class `GeometricObject` is implicitly a subclass of `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 (@), and the object’s memory address in hexadecimal. For example …

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

… displays something like `Loan@15037e5`.

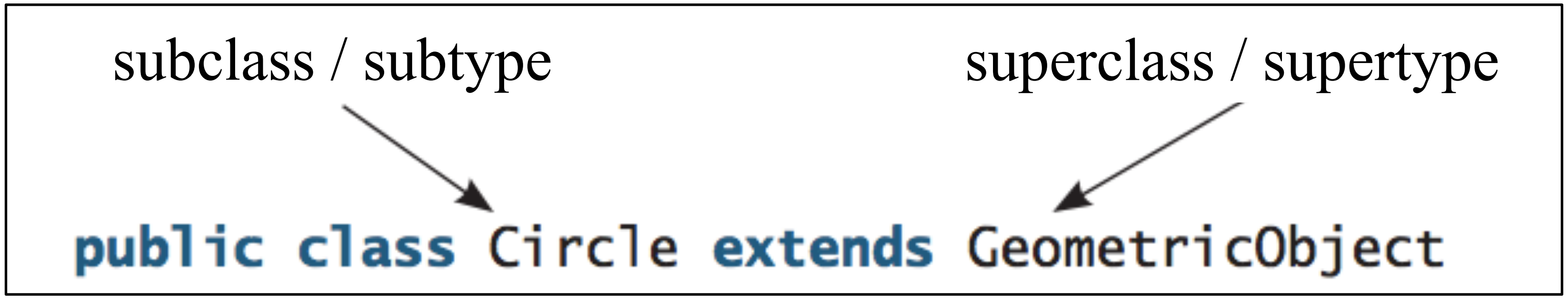
This message is not very informative… Thus, the `toString` method is usually overridden so that it returns a descriptive string representation of the object.

Polymorphism

To begin, some terminology…

A class defines a type. A type defined by a subclass is called a *subtype*, and a type defined by its superclass is called a *supertype*.

Example:



```
public class Circle extends GeometricObject
```

- `Circle` is a *subtype* of `GeometricObject`
- `GeometricObject` is a *supertype* for `Circle`

Polymorphism

The three pillars of OOP are *encapsulation*, *inheritance*, and *polymorphism*.

Polymorphism

The three pillars of OOP are *encapsulation*, *inheritance*, and *polymorphism*.

Polymorphism: a variable of a *supertype* can refer to a *subtype* object.

Polymorphism

The three pillars of OOP are *encapsulation*, *inheritance*, and *polymorphism*.

Polymorphism: a variable of a *supertype* can refer to a *subtype* object.

The inheritance relationship enables a subclass to inherit features from its superclass and define additional new features.

A subclass is a specialisation of its superclass:

- Thus, every instance of a subclass is also an instance of its superclass, but not vice versa.
- E.g. every circle is a geometric object, but not every geometric object is a circle.

Polymorphism

The three pillars of OOP are *encapsulation*, *inheritance*, and *polymorphism*.

Polymorphism: a variable of a *supertype* can refer to a *subtype* object.

The inheritance relationship enables a subclass to inherit features from its superclass and define additional new features.

A subclass is a specialisation of its superclass:

- Thus, every instance of a subclass is also an instance of its superclass, but not vice versa.
- E.g. every circle is a geometric object, but not every geometric object is a circle.

So we can write the following:

Polymorphism

The three pillars of OOP are *encapsulation*, *inheritance*, and *polymorphism*.

Polymorphism: a variable of a *supertype* can refer to a *subtype* object.

The inheritance relationship enables a subclass to inherit features from its superclass and define additional new features.

A subclass is a specialisation of its superclass:

- Thus, every instance of a subclass is also an instance of its superclass, but not vice versa.
- E.g. every circle is a geometric object, but not every geometric object is a circle.

So we can write the following:

- `Circle c = new Circle(5.0);`

Polymorphism

The three pillars of OOP are *encapsulation*, *inheritance*, and *polymorphism*.

Polymorphism: a variable of a *supertype* can refer to a *subtype* object.

The inheritance relationship enables a subclass to inherit features from its superclass and define additional new features.

A subclass is a specialisation of its superclass:

- Thus, every instance of a subclass is also an instance of its superclass, but not vice versa.
- E.g. every circle is a geometric object, but not every geometric object is a circle.

So we can write the following:

- `Circle c = new Circle(5.0);`
- `GeometricObject g = new Circle(5.0);`

Polymorphism

The three pillars of OOP are *encapsulation*, *inheritance*, and *polymorphism*.

Polymorphism: a variable of a *supertype* can refer to a *subtype* object.

The inheritance relationship enables a subclass to inherit features from its superclass and define additional new features.

A subclass is a specialisation of its superclass:

- Thus, every instance of a subclass is also an instance of its superclass, but not vice versa.
- E.g. every circle is a geometric object, but not every geometric object is a circle.

So we can write the following:

- `Circle c = new Circle(5.0);`
- `GeometricObject g = new Circle(5.0);`
- `Object o = new Circle(5.0);` `//` Why? – because `Object` is implicitly `//` a supertype for `GeometricObject`

Polymorphism

The three pillars of OOP are *encapsulation*, *inheritance*, and *polymorphism*.

Polymorphism: a variable of a *supertype* can refer to a *subtype* object.

The inheritance relationship enables a subclass to inherit features from its superclass and define additional new features.

A subclass is a specialisation of its superclass:

- Thus, every instance of a subclass is also an instance of its superclass, but not vice versa.
- E.g. every circle is a geometric object, but not every geometric object is a circle.

So we can write the following:

- `Circle c = new Circle(5.0);`
- `GeometricObject g = new Circle(5.0);`
- `Object o = new Circle(5.0);` `//` Why? – because `Object` is implicitly
  `//` a supertype for `GeometricObject`

But we cannot write the following:

- `Circle c = new GeometricObject();`

Dynamic Binding

Suppose the `toString` method defined in the `Object` class is overridden in the `GeometricObject` class.

Consider the following code:

```
Object o = new GeometricObject();

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

Which `toString` method is invoked by `o`?

Terminology:

- The *declared type* of `o` is `Object`
- The *actual type* of `o` is `GeometricObject`

In general, a method can be implemented in several classes along the inheritance chain.

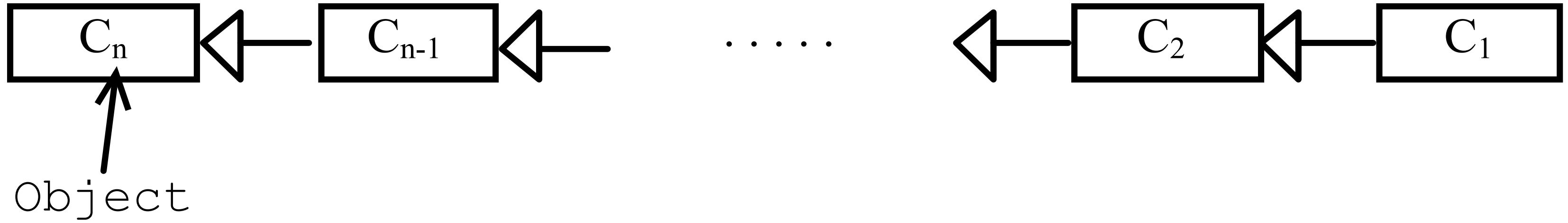
Based on the *actual type* of `o`, the JVM decides which method is invoked at runtime – this is called *dynamic binding*.

Dynamic Binding

Suppose an object `o` is an instance of class C_1 :

```
Object o = new C1();
```

Further, suppose 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 :



Note the following:

- C_n is the most general class. In Java, this is the `Object` class.
- Since `o` is an instance of C_1, it is also an instance of C_2, C_3, ..., C_{n-1}, C_n

If `o` invokes a method `foo`, the JVM searches for an implementation of `foo` 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.

Matching vs. Dynamic Binding

Matching a method signature and *binding* a method implementation are two separate issues…

Matching:

- The *declared type* of the object decides which method to match at compile time.
- The compiler finds a matching method according to the parameter type, number of parameters, and order of the parameters at compile time.

Binding:

- A method may be implemented in several classes along the inheritance chain.
- The JVM dynamically binds the implementation of the method at runtime, decided by the *actual type* of the object.

```java
public class DynamicBindingDemo2 {
   public static void main(String[] args) {
      Object g = new GraduateStudent(); foo(g);
      Object s = new Student(); foo(s);
      Object p = new Person(); foo(p);
      Object o = new Object(); foo(o);
   }

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

```java
public class GraduateStudent extends Student {
}
```

```java
public class Student extends Person {
   @Override
   public String toString() {
      return "Student";
   }
}
```

```java
public class Person {
   @Override
   public String toString() {
      return "Person";
   }
}
```

Method `foo` takes a parameter of the `Object` type. You can invoke `foo` with any object (because all objects are instances of `Object`). In general, an instance of a subclass can always be passed to a parameter of its superclass type.

```
public class DynamicBindingDemo2 {
   public static void main(String[] args) {
      Object g = new GraduateStudent(); foo(g);
      Object s = new Student(); foo(s);
      Object p = new Person(); foo(p);
      Object o = new Object(); foo(o);
   }

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

```
public class GraduateStudent extends Student {
}
```

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

```
public class Person {
   @Override
   public String toString() {
      return "Person";
   }
}
```

Method `foo` takes a parameter of the `Object` type. You can invoke `foo` with any object (because all objects are instances of `Object`). In general, an instance of a subclass can always be passed to a parameter of its superclass type.

Matching: the *declared type* of the object decides which method to match at compile time.

```
public class DynamicBindingDemo2 {
   public static void main(String[] args) {
      Object g = new GraduateStudent(); foo(g);
      Object s = new Student(); foo(s);
      Object p = new Person(); foo(p);
      Object o = new Object(); foo(o);
   }

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

```
public class GraduateStudent extends Student {
}
```

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

```
public class Person {
   @Override
   public String toString() {
      return "Person";
   }
}
```

Method `foo` takes a parameter of the `Object` type. You can invoke `foo` with any object (because all objects are instances of `Object`). In general, an instance of a subclass can always be passed to a parameter of its superclass type.

Matching: the *declared type* of the object decides which method to match at compile time.

Binding: the JVM dynamically binds the implementation of the method at runtime, decided by the *actual type* of the object.

```
public class DynamicBindingDemo2 {
   public static void main(String[] args) {
      Object g = new GraduateStudent(); foo(g);
      Object s = new Student(); foo(s);
      Object p = new Person(); foo(p);
      Object o = new Object(); foo(o);
   }

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

```
public class GraduateStudent extends Student {
}
```

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

```
public class Person {
   @Override
   public String toString() {
      return "Person";
   }
}
```

Method `foo` takes a parameter of the `Object` type. You can invoke `foo` with any object (because all objects are instances of `Object`). In general, an instance of a subclass can always be passed to a parameter of its superclass type.

Matching: the *declared type* of the object decides which method to match at compile time.

Binding: the JVM dynamically binds the implementation of the method at runtime, decided by the *actual type* of the object.

Output:

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

```
public class DynamicBindingDemo2 {
   public static void main(String[] args) {
      Object g = new GraduateStudent(); foo(g);
      Object s = new Student(); foo(s);
      Object p = new Person(); foo(p);
      Object o = new Object(); foo(o);
   }

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

```
public class GraduateStudent extends Student {
}
```

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

```
public class Person {
   @Override
   public String toString() {
      return "Person";
   }
}
```

Static vs. Dynamic Binding

When the *actual type* of the object **can** be determined at compile time, static binding is used (`private`, `static` and `final` methods – these methods cannot be overridden).

When the *actual type* of the object **cannot** be determined at compile time, dynamic binding is used (overridden methods)

Static binding occurs at compile time while dynamic binding occurs at runtime. Dynamic binding introduces a cost since the binding is performed at runtime.

Casting Objects

The following statement is an example of *implicit casting* – it is legal because an instance of `Circle` is an instance of `Object`:

```
Object o = new Circle();
```

However, the following statement results in a compilation error, even though `o` is actually a `Circle` object:

```
Circle c = o ;
```

Why? The reason is that a `Circle` object is always an instance of `Object`, but an `Object` is not necessarily an instance of `Circle`. In such cases, need to use *explicit casting*:

```
Circle c = (Circle)o;
```

What would happen if `o` was not actually an instance of `Circle`?

- A runtime error would occur.
- To ensure that an object is actually an instance of another object before attempting a casting, use the `instanceof` operator.

The `instanceof` Operator

Use the `instanceof` operator to test whether an object is an instance of a class:

```
Object o = new Circle();

// Perform casting only if o is an instance of Circle
if (o instanceof Circle)
  System.out.println("The diameter is " +
    ((Circle)o).getDiameter());
```

The `instanceof` Operator

Use the `instanceof` operator to test whether an object is an instance of a class:

```
Object o = new Circle();

// Perform casting only if o is an instance of Circle
if (o instanceof Circle)
  System.out.println("The diameter is " +
    ((Circle)o).getDiameter());
```

Note the following would result in an error because the member access operator (`.`) precedes the casting operator:

```
System.out.println("The diameter is " +
    (Circle)o.getDiameter());
```

Example: Demonstrating Polymorphism and Casting

The first example demonstrates the `instanceof` operator.

The second example creates two geometric objects: a circle and a rectangle. It then invokes a method named `display` to display the objects:

- The method displays the object's color and whether it is filled or not.
- The method also displays the area and diameter if the object is a circle, and displays the area and perimeter if the object is a rectangle.
- This example shows how polymorphism allows methods to be used generically for a range of object arguments. This is an example of *generic programming*.

PolymorphismAndCastingDemo1

PolymorphismAndCastingDemo2

The `equals` Method in `Object`

Like the `toString` method, the `equals` method is another useful method defined in the `Object` class:

```
public boolean equals(Object o)
```

This method tests whether two objects are equal. The syntax for invoking it is:

```
object1.equals(object2);
```

The default implementation of the `equals` method in the `Object` class is:

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

This implementation checks whether two reference variables point to the same object using the == operator.

You should override this method in your classes to test whether two objects have the same *content*.

Overriding the `equals` Method

The `equals` method in the `Circle` class can be overridden to compare whether two circles are equal based on, for example, the values of their radii as follows:

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

Overriding the `equals` Method

The `equals` method in the `Circle` class can be overridden to compare whether two circles are equal based on, for example, the values of their radii as follows:

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

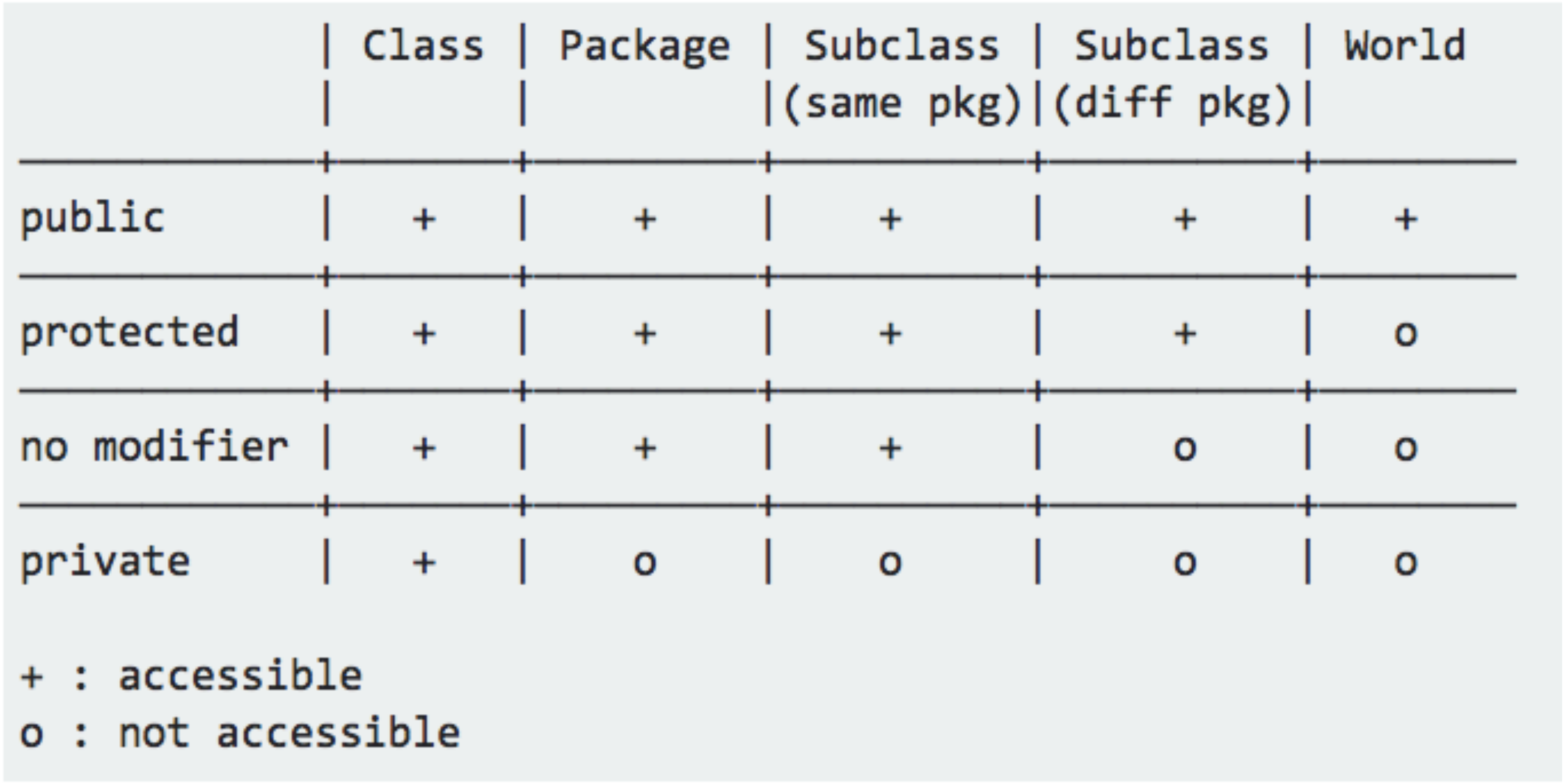
Note the type of the input argument – it is `Object`, not `Circle`.

Visibility Modifiers

So far we have used the `private`, `public`, and default visibility (aka accessibility) modifiers:

- If no visibility modifier is used, then by default the classes, methods, and data fields are accessible by any class in the same package (known as *package-private* or *package-access*).
- Use the `private` modifier to hide the members of the class completely so that they cannot be accessed directly from outside the class.
- Use the `public` modifier to enable the members of the class to be accessed by any class.
- Use the `protected` modifier to enable the members of the class to be accessed by classes in the same package or by the subclasses in any package.

Visibility Modifiers



```
            | Class | Package | Subclass | Subclass | World
            |       |         |(same pkg)|(diff pkg)|
────────────+───────+─────────+──────────+──────────+────────
public      |   +   |    +    |    +     |     +    |   +
────────────+───────+─────────+──────────+──────────+────────
protected   |   +   |    +    |    +     |     +    |   o
────────────+───────+─────────+──────────+──────────+────────
no modifier |   +   |    +    |    +     |     o    |   o
────────────+───────+─────────+──────────+──────────+────────
private     |   +   |    o    |    o     |     o    |   o

+ : accessible
o : not accessible
```

The `final` Modifier

Different uses of the final modifier:

1. A `final` class cannot be extended; for example:

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

2. A `final` method cannot be overridden by its subclasses.

3. A `final` variable is a constant; for example:

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

This Lecture…

In this lecture, we considered inheritance and polymorphism

The three pillars of object-oriented programming are: *encapsulation*, *inheritance*, and *polymorphism*.

Next lecture: abstract classes and interfaces.