### **Reading**

Please read Chapter 11 of book.

### **Inheritance**

Ability to define a new class from an existing class is called **Inheritance**.

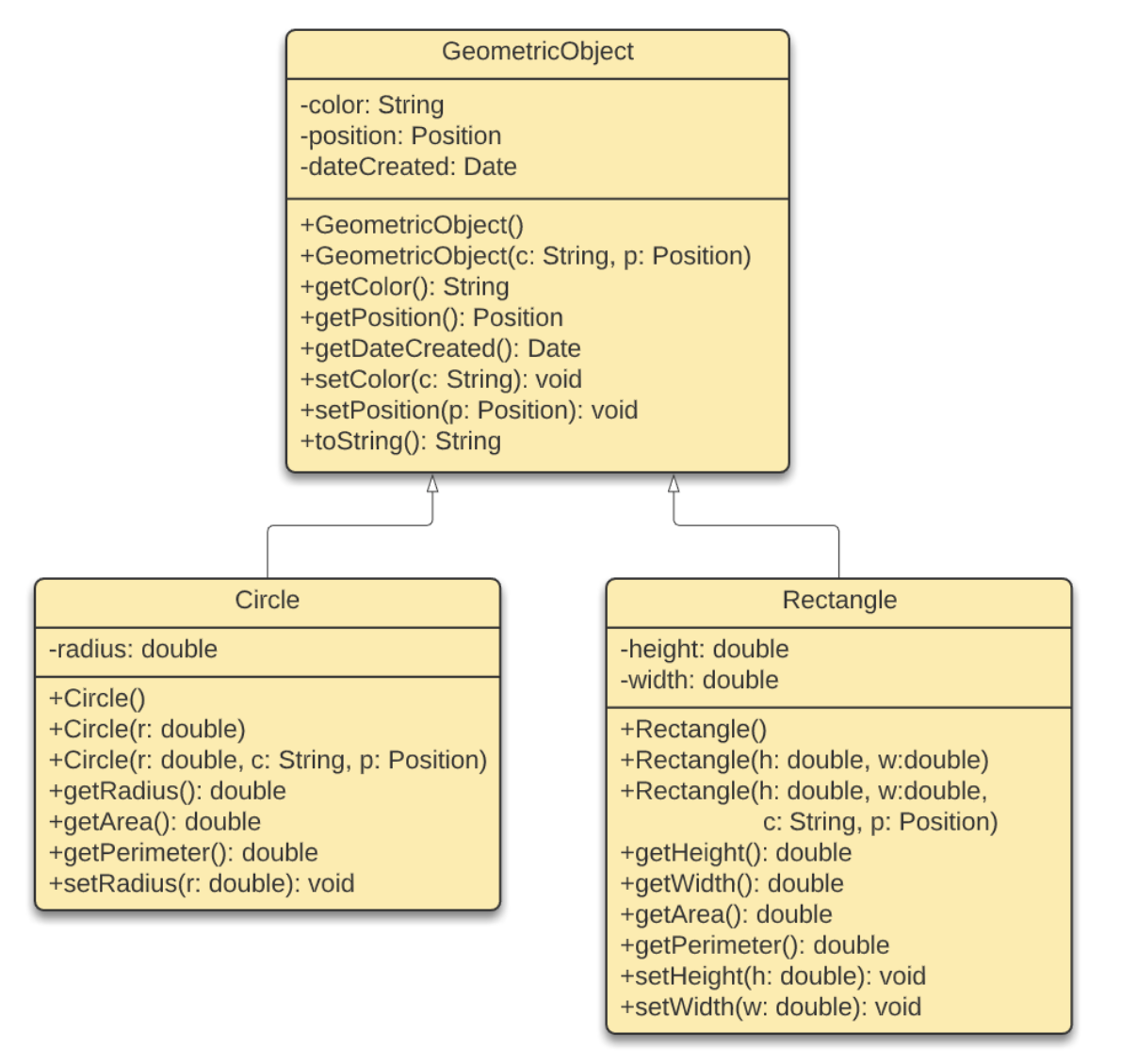
Can create a general class, called **superclass**, and later extend it to a more specialized class called a **subclass**.

Eg. a general GeometricObject class can be extended to a Circle class (or a Rectangle class).

Superclass GeometricObject will have common properties/behaviors (such as position, color, etc.).

Subclass Circle will be an extension of GeometricObject.

* It will would inherit all the properties/behaviors of GeometricObject.
* We could add properties (eg. radius) and behaviors (eg. getArea()) that are specific to Circles
* This is represented UML as below



Note the triangular arrow pointing to the superclass.

In Java, this relationship is represented with the **extends** keyword as below.

public class GeometricObject {

private String color;

private Position position;

...

// Constructors

public GeometricObject() {

dateCreated = new java.util.Date();

}

...

// Methods

public getColor() {

return color;

}

...

public setColor(String c) {

color = c;

}

...

}

public class Circle **extends** GeometricObject {

private radius;

...

// Constructors

public Circle() {

this(1.0);

}

public Circle(double r) {

radius = r;

}

public Circle(double r, String c, Position p) {

radius = r;

**setColor(c);**

...

}

...

// Methods

public double getArea() {

return radius \* radius \* Math.PI;

}

...

}

Keyword extends tells the compiler that Circle inherits from GeometricObject.

The constructor in Circle can use the method setColor() from its superclass.

Question: Can you replace the call "setColor(c);" with simply "this.color = c;"?

Inheritance models an **is-a** relationship. Eg. Circle is a GeometricObject.

In Java, a class can only inherit directly from one superclass (know as **single inheritance**).

### **Keyword "super"**

The keyword super is used to refer to the superclass.

Used to refer to the superclass methods or superclass constructors (which are not inherited).

Can call superclass constructor as follows:

public class Circle **extends** GeometricObject {

...

// Constructors

...

public Circle(double r, String c, Position p) {

**super(c, p);**

radius = r;

}

...

}

The call to superclass constructer must be the first statement in the subclasses constructor.

Even if no super class constructor is invoked, the compiler will automatically put "super()" as the first statement in the constructor.

So,

public Circle() {

radius = 1;

}

is the same as

public Circle() {

**super();**

radius = 1;

}

So a constructor will in turn call the no-arg constructors of the superclasses along its inheritance chain. This is called **constructor chaining**.

Because of this, you should always provide a no-arg constructors. This is to avoid compilation errors if you later extend the class.

Lastly, you can use the keyword super to invoke a superclass method as "super.getColor()".

### **Method Overriding**

A subclass can redefine a method that was defined in the superclass.

This method uses the same signature and return type as the method in the superclass. This is called **method overriding.**

public class GeometricObject {

...

**public String toString()** {

return "created on="+ dateCreated + ", color=" + color;

}

...

}

public class Circle extends GeometricObject {

...

**public String toString()** {

return "Circle {" + super.toString() + ", " + "radius=" + radius + "}";

}

...

}

We know of **Overloading**, which is defining several methods with the same name.

**Overriding** is proving a new implementation for a method in a subclass with the same (1) signature and (2) return type.

Aside: Like toString(), the equals(Object) is another useful method to override. Use it to test whether the objects have the same contents. Note that "==" only checks if two variables refer to the same object in memory.

### **Polymorphism**

Three pillars of object oriented programming are:

* Encapsulation
* Inheritance
* Polymorphism

Polymorphism refers to the ability of an object to take many forms.

Specifically, in Java, it is when a superclass is used to refer to a subclass object.

Consider this example.

public class TestProgram {

public static void main(String[] args) {

Circle circle1 = new Circle(10);

Circle circle2 = new Circle(20);

showObject(**circle1**);

showObject(**circle2**);

}

public static void showObject(**GeometricObject** obj) {

System.out.printin("Geometric Object: created on: " + obj.getDateCreated() + ", color: " + obj.getColor());

}

}

An object of subclass type can be used where ever its superclass is used.

Every Circle is also a GeometricObject. But every GeometricObject (eg. Rectangle) may not be a Circle.

### **Dynamic Binding**

Consider the code:

GeometricObject obj = new Circle();

System.out.println(obj.toString());

Which toString() will be called? The one for GeometricObject or the one for Circle?

In Java, the Java Virtual Machine (JVM) determines this dynamically.

It searches for an implementation of toString() in the hierarchy of classes starting as the most specific (Circle) to the least specific (GeometricObject).

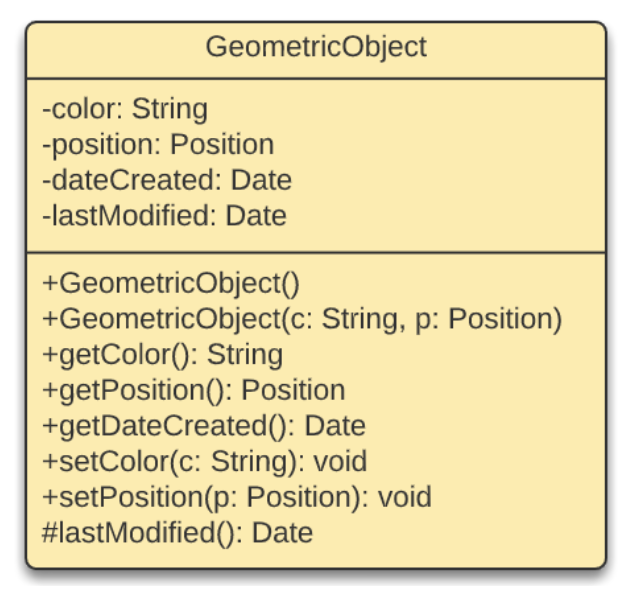
Once the implementation is founds, the search stops and that implementation is invoked.

In the above, the implementation in Circle is used.

### **Protected Variables and Method**

Use the visibility modified **protected** for variables and methods to be only accessible in the same class or subclasses.

In UML this is represented with is "#".



In Java this is represented below.

public class GeometricObject {

...

private Date lastModified;

...

**protected** Date lastModified() {

return lastModified;

}

...

}

Here subclasses have access to lastModified() but not any other class.

### **Preventing Extending and Overriding**

We have used the keyword **final** in variable declaration to represent constants.

We can prevent a class from being extended by using the final modifier.

public **final** class Circle {

...

}

Similarly, we can prevent a method from being overridden in subclasses by using the final modifier.

public class GeometricObject {

...

public **final** setColor(String c) {

...

}

}