COMP10062: Week 9 Guide

Sam Scott, Mohawk College, 2021

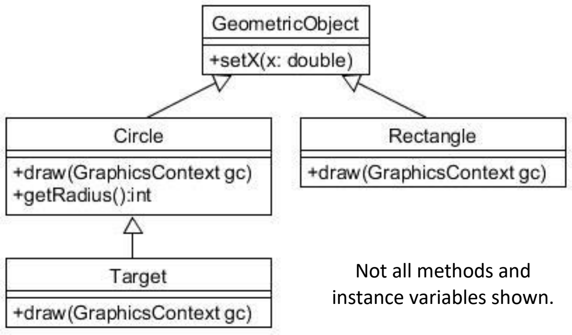
0 Reading for this Week

For this week, you should read **sections 8.3 and 8.4** (skip the sections on the extending interfaces and the `Comparable` Interface).

1. Polymorphism (Read Section 8.3)

A class defines a **type** and can be used to create objects.

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


GeometricObject
+setX(x: double)
Circle
+draw(GraphicsContext gc)
+getRadius():int
Rectangle
+draw(GraphicsContext gc)
Target
+draw(GraphicsContext gc)

Not all methods and instance variables shown.

Classes can have **subtypes** (Classes that extend them) and **supertypes** (classes that they extend.)

A variable of type A can hold references to objects of type A or any subtype of type A.

```
GeometricObject c = new Circle();
```

In the above example, `c` (the variable) is of type `GeometricObject` but the object it references is of the subtype `Circle`.

The **declared type** is the type the variable was given when it was declared. The **actual type** is the type of the object stored in the variable.

You can only directly access the fields and methods of an object's declared type.

```
c.setX(10);                           // this works
System.out.println( c.getRadius() );  // this is a syntax error.
```

If you want to access the fields and methods of an object's actual type, you have to cast it, like this:

```
Circle z = (Circle) c;
System.out.println( z.getRadius() );
```

Or like this:

```
System.out.println( ((Circle) c).getRadius() );
```

Gotcha

If you cast an object incorrectly (e.g. it's a `Rectangle` and you cast it as a `Circle`) you risk getting an exception. The fix is to make sure you know what type it is before you cast it.

If you're not sure what the actual type of an object is (maybe your method received a parameter of type `GeometricObject`) you can find out using the boolean `instanceof` operator.

```
if (x instanceof Circle)
    System.out.println( ((Circle) x).getRadius() );
```

Which Method Gets Called?

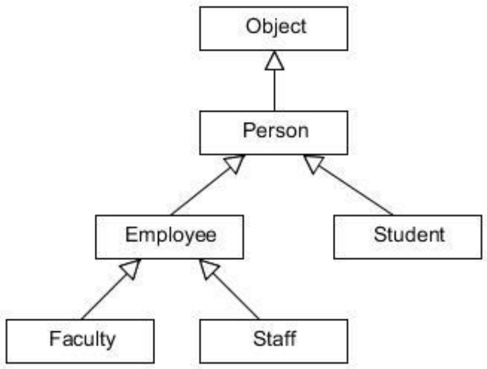
Suppose you have a class named `Circle` with a subclass named `Target`. They both have a draw method.

```
Circle t = new Target();
t.draw();
```

The above code is legal because the declared type contains a `draw()` method. But it is the `draw()` method from the actual type (`Target`) which gets called.

What's the Point?

Suppose you had three different types of `Person` objects: `Faculty`, `Student` and `Staff`. And suppose they could all join the college health club but with different privileges and fees.



Methods

You need to write a static `billing` method that calculates and prints bills for health club members.

You could write three versions of the method, as shown below.

```
public static double billing(Faculty member) { ... }
public static double billing(Student member) { ... }
public static double billing(Staff member) { ... }
```

Or, you could write just one version, like this:

```
public static double billing(Person member) { ... }
```

This method can accept `Faculty`, `Student` or `Staff` objects.

Arrays

Similarly, suppose you need to store all of the members, of all different types. You could create three arrays, like this:

```
Faculty[] facultyMembers = new Faculty[100];
Student[] studentMembers = new Student[100];
Staff[] staffMembers = new Staff[100];
```

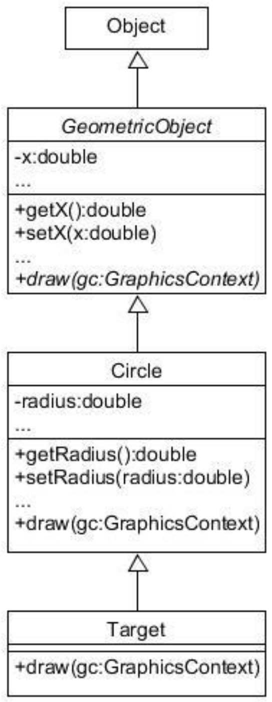
Or you could create a single array of type Person that could store objects of all 3 types.

```
Person[] members = new Person[100];
```

2. Abstract Classes and Methods (Read Section 8.4, pp. 646-648)

When designing a class hierarchy, superclasses should be more general and subclasses should be more specific (e.g. Fruit is more general than Apple, so Fruit should be the superclass).

Sometimes a class is so general that it cannot or should not be used to create an object. This is an **abstract class**, denoted by italics in class diagrams as shown at right.



Abstract classes are defined just like regular classes except:

1. They use the keyword `abstract` in the class header line.
2. Some of their methods can also be abstract.
3. They cannot be instantiated (used to create objects).

For example, `GeometricObject` is not specific enough to draw or compute an area and perimeter, so you probably shouldn't create objects of that type.

Try it Now

Get the `GeometricObject` code from Canvas and make the class abstract to match the class diagram above:

```
public abstract class GeometricObject { …
```

Abstract methods are just method headers. They have no bodies and use the keyword `abstract` in their method header line. In class diagrams, they also appear in *italics*. In the diagram on this page, the class name *GeometricObject* is in italics and the *draw* method in GeometricObject is also in italics.

Try it Now

Add an abstract `draw()` method to the `GeometricObject` class, so that it matches the class diagram above.

```
public abstract void draw(GraphicsContext gc);
```

Non-abstract subclasses like `Circle`, `Rectangle` and `Target` must implement these abstract methods. Making a class abstract can also be a way of forcing the subclasses to contain certain methods.

Try it Now

Go to the Draw.java class and change `((Circle) c)` to `c`. Why is this allowed now?

Try it Now

Get the sample code from Canvas. Create a new class that extends `GeometricObject`, but leave the class body blank. What Syntax errors do you get? How can you fix them?

In Draw.java, try to create a new `GeometricObject` object. What Syntax error do you get?

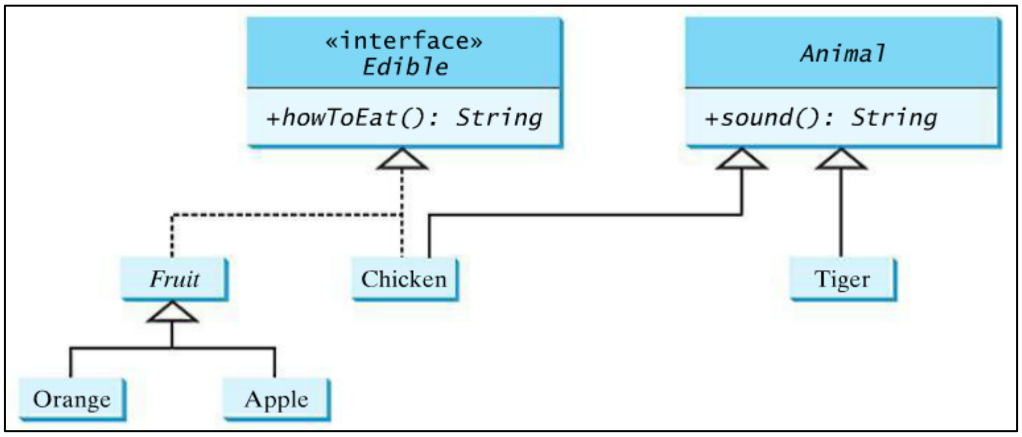
3. Interfaces (Read Section 6.4, pp. 621-628)

Interfaces are like abstract classes except:

1. They use the keyword `interface` in the class header line instead of the word `class`.
2. All of their methods have to be abstract.
3. All of their variables must be static, public and final (i.e. constants) and they must be given a value when they are declared.

Subclasses do not `extend` interfaces, they `implement` them. This is shown as a dotted inheritance arrow, as shown in the UML diagram below. Interfaces appear in italics, but usually with the tag <<*interface*>> beside the name.

The example class diagram below mirrors the example code on Canvas. In this picture, Everything in *Edible* and *Animal* is in italics. The class name *Fruit* is also in italics.



Example from p. 572 of Introduction to Java Programming, 10th Edition*, by Y. Daniel Liang.*

The declaration for the class Chicken in the above diagram looks like this:

```
public class Chicken extends Animal implements Edible { …
```

Interfaces are a contract. If you implement an interface, you are contractually obligated to implement the abstract methods it contains.

In the example UML diagram abaove, an object of type `Chicken` is also of type `Edible` and `Animal`. The dotted arrows indicate inheritance, but using `implements` rather than `extends`. The "implements" relationship is an "is a" relationship, just like inheritance. So in the above example, a `Chicken` is an `Animal` and an `Edible`, an `Orange` is a `Fruit` and an `Edible`, etc.

A class can only `extend` one other class, but it can `implement` as many as you like using a comma-separated list. So interfaces provide a type of **multiple inheritance**. For example, if you also defined a Bird interface, you could have Chicken implement it as well, like this:

```
public class Chicken extends Animal implements Edible, Bird { …
```

4. The Comparable Interface (Optional Extra – pp. 642-646)

You'll find interfaces all over the Java and Java FX APIs. Here's just one example.

`Arrays.sort()` can be used to sort an array, and `Arrays.binarySearch()` can be used to efficiently search a sorted array. These methods requires a parameter of type `Comparable[]`. Comparable is an interface that forces the implementation of a `compareTo()` method. Whenever these methods have to compare two elements in the array, they do it by calling a `compareTo()` method.

These methods work for arrays of Strings because the `String` class implements the `Comparable` interface. If you have a class that implements `Comparable`, theses methods will work for your class as well. Here's what the `Comparable` interface looks like in code:

```
public interface Comparable<E> {
      public abstract int compareTo(E o);
}
```

If you implement this interface, you must have a `compareTo()` method which compares the current object to the object passed as a parameter. This method returns a negative, zero or positive integer depending on whether your object is less than, equal to, or greater than the one passed as a parameter, respectively.

When you implement `Comparable` you have to replace every occurrence of `E` with the name of your class, like this:[1]

```
public class MyClass implements Comparable<MyClass> {
      ...
      public int compareTo(MyClass other) {
            // compare this to other here, and return the result.
      }
      ...
}
```

Then if you have an array of type `MyClass`, it is also an array of type `Comparable` and you can sort it using `Arrays.sort()`.

Note that the `Comparable` interface forces you to implement this method, but it's completely up to you how you do the comparison. For example if the class defines a Person, you could compare them by their first name, last name, age or some combination of the three. It all depends on what kind of sorting you want to do.

See the `Num` class on Canvas for an example.

[1] This is something called a "generic". You'll learn more about this in your .NET and Data Structures and Algorithms courses.