COMP10062: Week 8 Guide

Sam Scott, Mohawk College, 2020

0 Reading for this Week

For this week, you should read **sections 8.1 and 8.2**.

1. Inheritance (Read section 8.1, pp. 586-592)

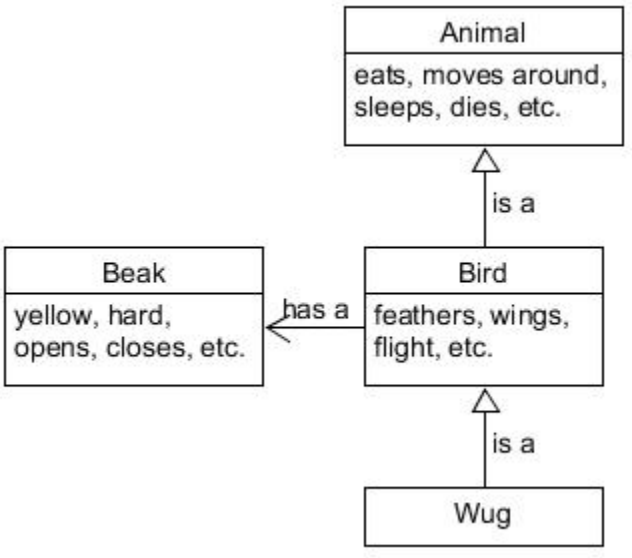
Inheritance is the second important relationship between classes. Like association, inheritance mirrors an important way in which human beings understand and organize knowledge of the world.

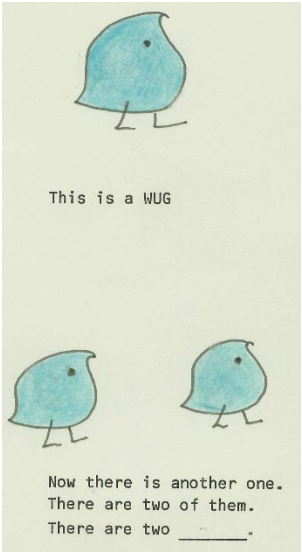
For example:

- A bird *has a* beak.
  - This is an **association** between the concepts BIRD and BEAK.
- A bird *is an* animal.
  - This is an **inheritance** relationship between the concepts BIRD and ANIMAL. Birds are a kind of animal. They are a **subclass** of the class of animals.
- An animal eats.
  - This is a property of the concept ANIMAL.
  - Because animals eat, birds also eat. This property is INHERITED.

Inheritance is a powerful tool. If I tell you that a wug is a kind of bird, you can immediately tell me a lot about wugs. You know that they have beaks, that they are animals, that they eat, and lots more. This information is all inherited from the BIRD and ANIMAL concepts.

In computer science, we can set up inheritance relationships between classes to eliminate code duplication and modularize our code even further.





P.S. "Wug" comes from Jean Burko Gleason's "wug test", a famous study in linguistics to investigate how children acquire the rules of language. https://en.wikipedia.org/wiki/Jean_Berko_Gleason

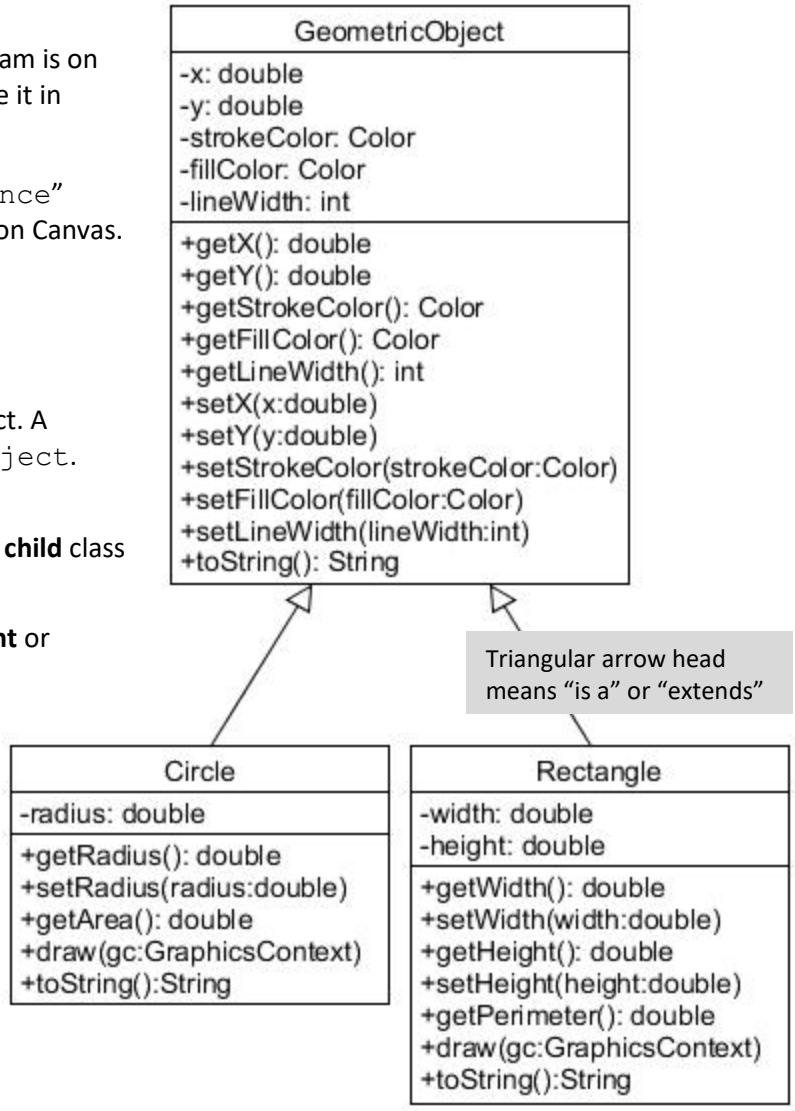
Java Inheritance Example

The code that matches this class diagram is on Canvas. Run the **Draw.java** class to see it in action.

Compare to the "`BeforeInheritance`" versions of these classes, also posted on Canvas.

Notice all the duplicated code.

Inheritance reduces code duplication.



- A `Circle` **is a** `Geometric` object. A `Rectangle` **is a** `GeometricObject`.
- The `Circle` class **extends** the `GeometricObject` class. It is a **child** class or a **subclass**.
- `GeometricObject` is the **parent** or **superclass** or **base** class.
- A `Circle` object contains all methods and fields (instance variables) in the `Circle` and `GeometricObject` classes.
- But `Circle` class methods only have access to public members of `GeometricObject`.
- So subclasses have to use getters and setters to access `x`, `y`, `lineWidth`, etc.
- The `toString()` methods in `Circle` and `Rectangle` **override** the `toString()` method in `GeometricObject`. Note the use of the optional `@override` tag in the code.

Issues…

In `Circle` and `Rectangle`, it would be great to call the `toString()` method from `GeometricObject`, to output the private variables stored there. See section 2 below.

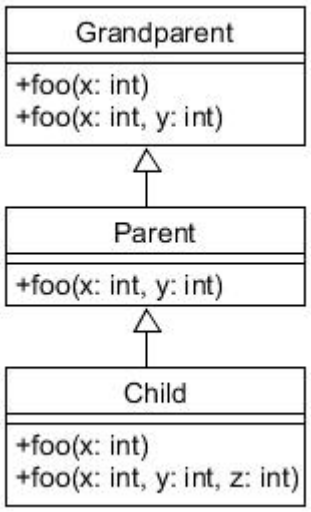
Why do none of these classes have constructors? Because constructors are more complicated and are not inherited. We'll add constructors in section 3 below.

2. Overriding Methods (Read Section 8.1, pp. 592-598)

Overriding and overloading are two different ways in which an object could contain two methods of the same name.

Overriding = exact same method signature in a superclass and a subclass.

Overloading = methods with different signatures in the same class or split between superclass and subclass.



Quick Exercises

Consider the UML diagram on the right.

1. Which of the methods named foo in the UML diagram on the right could be considered overrides? Which are better thought of as overloads? Which could be considered both?

2. Consider the following code:

```
Child c = new Child();
Parent p = new Parent();
Grandparent g = new Grandparent();
```

   How many methods with the name "foo" does each of the objects `c`, `p` and `g` contain? Which ones can you call from object `g`?

3. For each line of code below, indicate on the UML diagram which method will get called. Watch out – some of these might be syntax errors!

```
1. c.foo(1);
2. c.foo(1, 2);
3. c.foo(1, 2, 3);
4. p.foo(4);
5. p.foo(4, 5);
6. p.foo(4, 5, 6);
7. g.foo(7);
8. g.foo(7, 8);
9. g.foo(7, 8, 9);
```

   Run the **OverrideTests.java** app to see if you are right.

The super Keyword

If you want to force a call to a method from a superclass, you can use the `super` keyword.

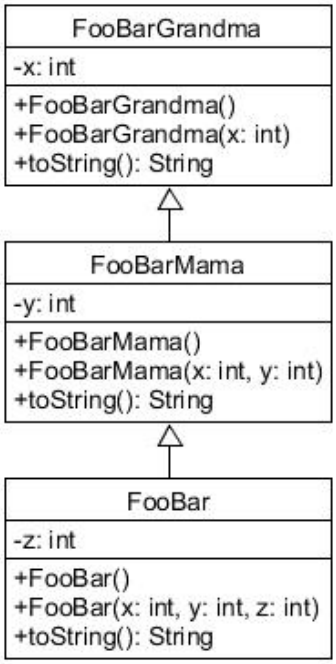
The `super` keyword is a bit like the `this` keyword, but it forces Java to call the version of the method in the superclass instead of in the current class.

`this.foo()` will call the `foo()` method accessible to the current class.

`super.foo()` will call the `foo()` method accessible to the superclass of the current class.

3. Constructors and Inheritance (Read section 8.2, pp. 599-611)

Consider the UML class diagram on the right. Make sure you understand it and that you can identify the constructors for each class. The code for these classes is on Canvas.



Important Reminders

- Constructors are called once when an object is created and never again for that object.
- If you don't write a constructor, a default constructor is supplied that has no parameters and does nothing.

Constructor Inheritance

- Constructors are **not inherited**.
- When an object is created, **one constructor will be called for every class** in its inheritance hierarchy.
- By default, Java will call the constructor you asked for, and then the no-argument constructor for every superclass.

Run **FooBarTest.java** on Canvas to see which constructors get called when you invoke either of the `FooBar` constructors.

Notice that the `y` and `z` values don't get set properly in all cases. We'll resolve that in a moment.

Quick Exercise

What will happen if you remove the no-argument constructor from FooBarGrandma or FooBarMama?

Try commenting one of them out in IntelliJ and saving the class. What happens? Can you explain it?

The `super` Keyword

A constructor is typically used to set the values of all or most instance variables. `FooBar` has a 3-argument constructor because a `FooBar` object has 3 instance variables (*what are they?*). But right now, this constructor only sets the value for `x` because it doesn't have access to the other two (*why not?*). The obvious thing to do is to call the `FooBarMama` constructor to set the other two.

- Recall: The `this()` keyword can be used as the first line of a constructor to call another constructor from the same class.
- Similarly, the `super()` keyword can be used to call a constructor from the superclass.

If you don't include an explicit call to a superclass constructor, Java inserts `super();` for you.

Quick Exercise

Use the `super` keyword to fix `FooBar` and `FooBarMama` so that all instance variables get set when the constructors that take arguments get called.

Golden Rule of Constructor Initialization

Initial values for instance variables should be set by constructors in the same class.

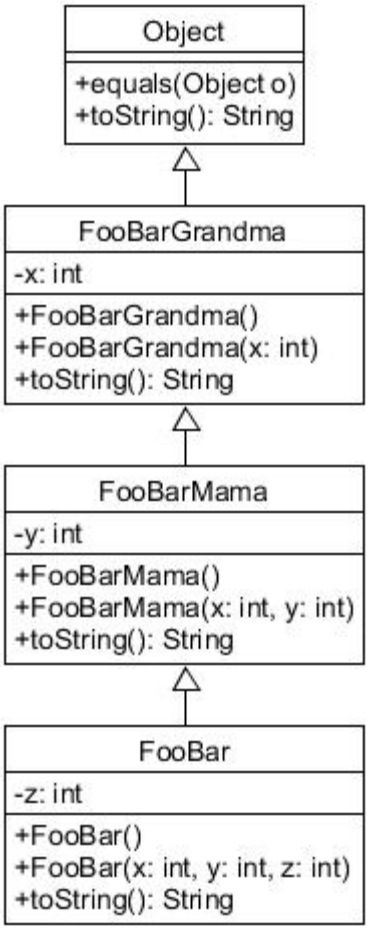
4. The Object Class (Read Section 8.2, pp. 612-616)

Every class that does not have an "extends" clause in its definition extends `Object` by default.

This means that **every class is a descendent of the Object class**. (*Why?*)

The `Object` class contains a few generic methods, such as:

```
.equals(Object 0)
.toString()
```



Now it all makes sense!

This is why you can print every object even without a `toString()` method.

- `System.out.println()` accepts an argument of type `Object`, then calls its `toString()` method. If the object's base class doesn't have a that method, it uses the one inherited from `Object`.
- A parameter of type `Object` can accept anything because every object of any class type is also an object of the `Object` type (more on this next week when we cover Polymorphism).

And it's why you can concatenate every object.

- The concatenation operator (+) also accepts an argument of type `Object` and calls its `toString()` method.

And it's why IntelliJ always includes an `@Override` tag when it generates a `toString()` method.

Finally, this also explains why the `setText()` method of Label and the `fillText()` method of `GraphicsContext` ***don't*** work for every object.

- These methods require `String` parameters. So you have to actually call `toString()` on your object if you want to use it in these methods.

And you probably didn't know this, but you can also always use `.equals()` to compare two objects, even if you don't define an `equals()` method – for the same reason that you can always use `.toString()`. The inherited `equals()` method just uses == to compare the two objects. (*Will it ever return true?*)