ALGONQUIN COLLEGE

MODULE 03: INHERITANCE AND ABSTRACTION

Professor : Dave Houtman

Office: T323

Office Hrs: Wednesday 14:15 – 15:30

Wednesday (after lecture*)

* confirm beforehand

Email: houtmad@algonquincollege.com

3.1 Relationships Between Objects

There are four kinds of relationships between objects. Thus far, you've been introduced to three of them in some form:

1. Association
2. Aggregation
3. Composition

Note: Aggregation and Composition are sometimes lumped together under the single heading 'Composition', since both are *has a* relationships. This was done in the previous chapter using the example of the relationship between Truck Factories, Trucks, and their components

3.1 Relationships Between Objects

We might graphically express these relationships as follows:

1. Association

A factory makes pickup trucks

A **composition** is another form of relation between classes that indicates a one-to-one ownership relationship between one object and another.

2. Aggregation

A pickup truck '*has*' wheels

3. Composition

A pickup truck '*has a*' steering wheel

3.1 Relationships Between Objects

In code, this would look like:

1. Association

```
TruckFactory.buildTruck();
```

2. Aggregation

```
public class Pickup {
    Wheel[] tires = new Wheel[4];
}
```

3. Composition

```
public class Pickup {
    SteeringWheel mySW;
                = new SteeringWheel;
```

3.1 Relationships Between Objects

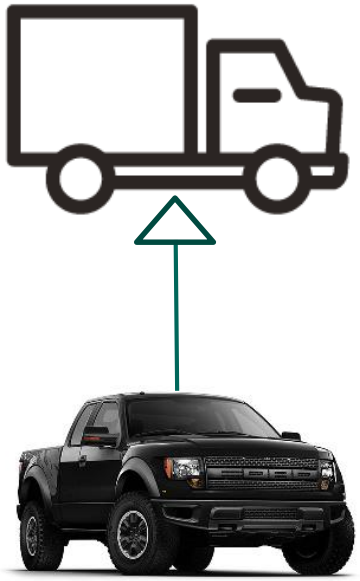
Inheritance constitutes the fourth kind of relationship between objects. Recall from Module 01 that ***inheritance*** allows a subclass to be derived from its parent superclass. The subclasses will inherit all of the non-private fields and methods (i.e. the ***members***) of the superclass. In addition to adding its own, new members, superclass members may be ***overridden*** in the subclass. Thus inheritance is a form of ***code reuse***, a central tenet of the Object-Oriented Programming (OOP) paradigm.

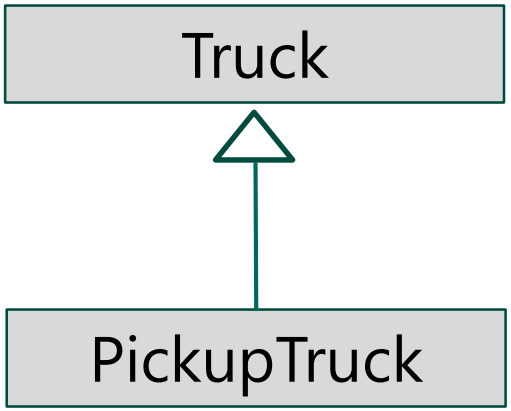
CookieCutter
superclass

SnowflakeCookie, GingerbreadManCookie, etc.
(subclasses)

3.2 Inheritance in UML Diagrams

Inheritance is indicated in UML diagrams using the open arrow notation. Note that the arrow points *from the subclass **to** the superclass*.



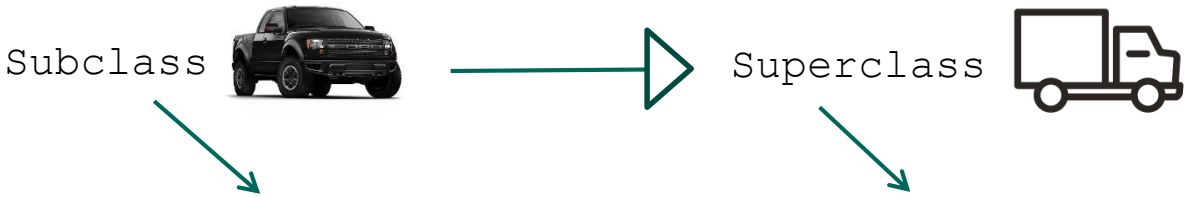


While aggregation and composition form '*has a*' relationships, inheritance forms an '*is a*' relationship. A pickup truck '*has a*' steering wheel and '*has*' tires, but it '*is a*' truck.

3.2 Inheritance in UML Diagrams

Inheritance leads to hierarchical relationships and code reuse. The superclass (also known as the ***base class***) provides members that the subclass (or ***derived class***) automatically inherits. (Thus even though the UML arrow points *up* to the superclass, the methods are passed *down* to the subclass.)

The Java keyword that indicates that a subclass is derived from a base/superclass is `extends`, used as follows:




```
public class DerivedClass extends BaseClass{
    // add new members here, or
    // override BaseClass members
}
```

3.2 Inheritance in UML Diagrams

For example, consider the following UML diagram for a base class called `GeometricObject`

GeometricObject
-colour: String -filled: boolean -dateCreated: java.util.Date
+GeometricObject() +GeometricObject(colour: String, filled: boolean) +setColour(colour: String): void +getColour(): String +isFilled(): boolean +setFilled(filled: boolean): void +getDateCreated(): java.util.Date +toString(): String

This example taken from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pp. 410-414.

3.3 Inheritance

The code corresponding to this UML class diagram looks like this:

```
public class GeometricObject
  private String colour = "white";
  private boolean filled;
  private java.util.Date dateCreated;

  public GeometricObject(){
    dateCreated = new java.util.Date();
  }

  public GeometricObject(String colour,
    boolean filled)
    dateCreated = new java.util.Date();
    this.colour = colour;
    this.filled = filled;
  }

  public void setColour(String colour){
    this.colour = colour;
  }
```

```
public String getColour(){
  return colour;
}

public boolean isFilled(){
  return filled;
}

public void
setFilled(boolean filled){
  this.filled = filled;
}

public java.util.Date
getDateCreated(){
  return dateCreated;
}

…etc.
```

3.3 Inheritance

If we derive a new class `Circle` using:

```
public class Circle extends GeometricObject{
    …
}
```

then the newly derived class inherits all of the members of the base class, excluding the superclass constructors and non-private members:

Liang 9 | D&D 9

3.3 Inheritance

Rather than repeat the obvious—we already know that every non-private method in `GeometricObject` will be found in the `Circle`—our UML diagram only needs to display the new members that will be added to the subclass. The members inherited from the superclass are assumed; we don't need to repeat them in the subclass.

Assume we wish to add the following members to the `Circle` class.

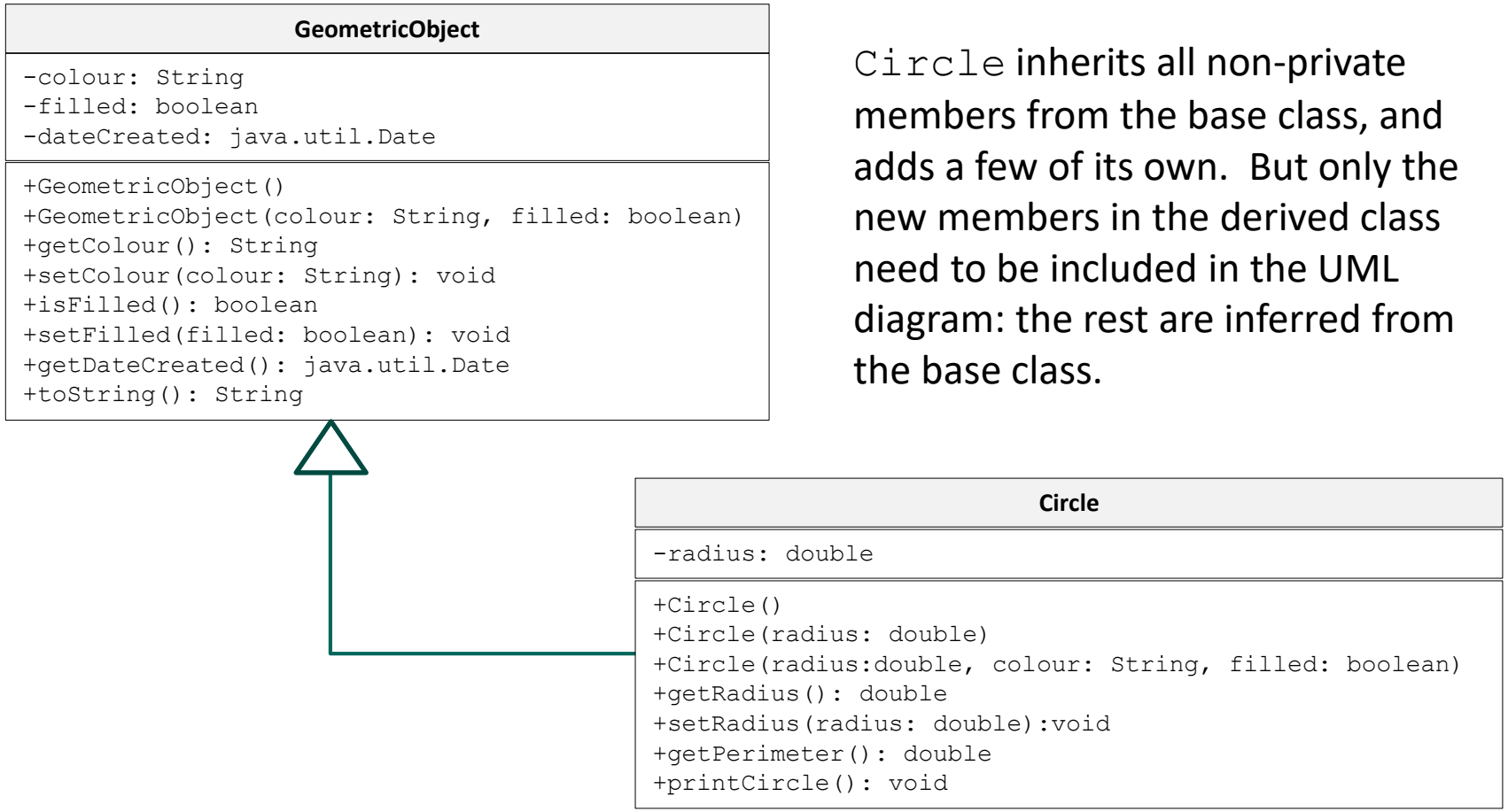
Circle
`-radius: double`

`+Circle()`
`+Circle(radius: double)`
`+Circle(radius:double, colour: String, filled: boolean)`
`+getRadius(): double`
`+setRadius(radius: double):void`
`+getArea(): double`
`+printCircle(): void`

Only the new members need to appear in the UML diagram for `Circle`.

3.3 Inheritance

Since the UML diagram need only reflect the new features of the derived class, the actual relationship will look like this:



`Circle` inherits all non-private members from the base class, and adds a few of its own. But only the new members in the derived class need to be included in the UML diagram: the rest are inferred from the base class.

3.3 Inheritance

The new code corresponding to the `Circle` UML class diagram looks like this:

```
public class Circle
                extends GeometricObject{
  private double radius;

  public Circle(){
  }

  public Circle(double radius)
    setRadius(radius);
  }

  public Circle(double radius, String
    colour, boolean filled){
    setRadius(radius);
    setColour(colour);
    setFilled(filled);
  }

  public void setRadius(double radius){
    this.radius = radius;
  }
```

```
public double getRadius(){
  return radius;
}

public double getPerimeter(){
  return 2 * radius * Math.PI
}

public void printCircle(){
  System.out.println("The
  circle was created on " +
  getDateCreated() + " and
  the radius is " + radius
}

…etc.
```

3.3 Inheritance

So `Circle` inherits all of `GeometricObject`'s non-private methods, and adds a few of its own. Note that one of our `Circle` constructors uses two superclass methods: `setColour()` and `setFilled()` are as accessible to `Circle` as they would be if they were defined in the `Circle` subclass itself.

```
public class Circle extends GeometricObject{
  private double radius;

  public Circle(){
  }

  //…etc.

  public Circle(double radius, String colour, boolean filled){
    setRadius(radius);
    setColour(colour);
    setFilled(filled);
  }

  public void setRadius(double radius){
    this.radius = radius;
  }
```

`setColour()` and `setFilled()` are declared in the superclass `GeometricObject`.

Note: you don't need to use `super` here; You inherit these automatically.

3.3 Inheritance – Q & A

Q. Can you inherit from every class?

A. No. Classes which are declared `final` cannot be extended or subclassed. One such example is the `Math` class. Since all its members are static, you can call each of its fields and methods using e.g. `Math.PI` or `Math.sin()`. There's no need to extend it, since it's already 'complete.'

Another example of a final class is the `String` class. The declaration of `String` begins*:

```
public final class String…
```

`final` makes each new `String` **immutable**; it says: you can't change this.

Another example of a final class?: `Scanner`.

*From: http://docs.oracle.com/javase/7/docs/api/java/lang/String.html. Retrieved October 27, 2015

Liang 11.15 | D&D 10.7

3.3 Inheritance – Q & A

Q. Can you inherit from more than one class?

A. In Java, no. Only ***single inheritance*** is permitted. In other OOP languages (like C++) ***multiple inheritance*** is allowed. However, in addition to classes, Java uses *interfaces*, which are similar to classes, but with restrictions. And you *can* use multiple interfaces. So you can achieve something like multiple inheritance.

3.3 Inheritance – Summary

Inheritance forms the fourth and final relationship between objects.

1. Association

A factory makes pickup trucks

4. Inheritance

A pickup truck *'is a'* truck

2. Aggregation

A pickup truck '*has*' wheels

3. Composition

A pickup truck '*has a*' steering wheel

3.3 Inheritance – Summary

The word `extends` tells you that (in this analogy) a `Pickup` *is a* `Truck`, and inherits all the features and behaviours associated with the `Truck` class.

1. Association

```
TruckFactory.buildTruck();
```

4. Inheritance

```
public class Pickup extends Truck {
  … }
```

2. Aggregation

```
public class Pickup {
   Wheel[] tires = new Wheel[4];
}
```

3. Composition

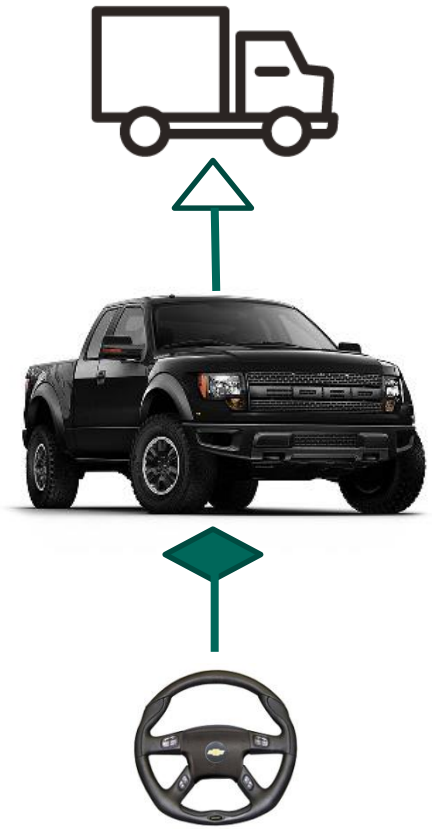
```
public class Pickup {
   SteeringWheel mySW;
            = new SteeringWheel;
```

3.3 Inheritance – Summary

In common programming parlance, one frequently ignores associations and aggregations, and speaks only of *is a* and *has a* relationships—a pickup *is a* truck and it *has a* steering wheel. In broad terms, objects are connected by inheritance and composition only; any other kind of relationship may be considered a variation on these two main themes..

Inheritance

Composition



3.4 The `protected` Access Modifier

As we described earlier, Java offers three primary types of access modifier:

`public`

- Members declared `public` are visible to all classes.
- Public members are automatically inherited in subclasses.
- Public classes are visible *across* packages.

`protected`

- Members declared `protected` are intermediate in their visibility to other classes. Protected members are public *inside* a package and normally private *outside* the package. But they are visible via inheritance, *even in when that inheritance is in an external package*.

`private`

- Members declared `private` are only visible within the class in which they are declared.
- Private members are not visible to outside classes, and they are not inherited in subclasses.

Liang 11.14 **D&D 9.3**

ALGONQUIN COLLEGE

3.4 The `protected` Access Modifier

Thus all variables declared `protected` within a package behave like `public` members. But outside that package, they behave like `private` members.

package green;

```
public class C1{
    public int x;
    protected int y;
    protected void m(){}
    private int z;
}
```

```
public class C3 {
    C1 c = new C1();
    can access c.x;
    can access c.y;
    can invoke c.m();
    cannot access c.z;
}
```

```
public class C2 extends C1{
    can access x;
    can access y;
    can invoke m();
    cannot access z;
}
```

```
public class C4 {
    C1 c = new C1();
    can access c.x;
    cannot access c.y;
    cannot invoke c.m();
    cannot access c.z;
}
```

package orange;

3.4 The `protected` Access Modifier

However, if a class with protected members is *extended* in another package, then those protected members are treated as public. So protected members enjoy special status; normally private outside a package, if inherited in a subclass they can tunnel across packages back to their base class.

package green;

```
public class C1{
    public int x;
    protected int y;
    protected void m(){}
    private int z;
}
```

```
public class C3 {
    C1 c = new C1();
    can access c.x;
    can access c.y;
    can invoke c.m();
    cannot access c.z;
}
```

```
public class C2 extends C1{
    can access x;
    can access y;
    can invoke m();
    cannot access z;
}
```

```
public class C4 extends C1 {
    can access x;
    can access y;
    can invoke m();
    cannot access c.z;
}
```

package orange;

3.5 Using `super`

As described earlier, when two or more methods have the *same* name and a *different* signature, they are considered to be *overloaded*. Both methods are part of the same class.

A reminder: the signature consists of:

1. The name of the method
2. The number of parameters
3. The type of each parameter
4. The order of the parameters

…and the signature does *not* include a method's return type, or the names of the identifiers used.

Liang 11.3

3.5 Using super

With overloading, we frequently used `this` to connect different overloaded methods of the same class. Recall the 'trick' that helps explain which method/constructor is being called whenever `this` appears in a statement

`this``.someFieldIdentifier`

- Substitute the Class name into `this`; that will be the instance variable used at run time. So, in the above example

`this.radius` "means" `Circle.radius`

`this``(someIdentifier1, someIdentifier2)`

- Substitute the `Class` name when you see `this`; therefore, `this` is being used to refer to a constructor, and the number of parameters passed indicate which constructor is being used (assuming there are overloaded constructors present)

3.5 Using super

Similarly, the word `super` refers to fields and methods in the *superclass* of the current class. The same trick applies, but this time you should conceptually apply the name of the superclass of the current object whenever you see the word `super`. Thus when you see:

`super.``someFieldIdentifier`

- Substitute the parent class's name into `super`; that will be the instance variable used at run time;
- `super` is commonly used inside methods, when an instance field in the parent class needs to be used in the subclass. If you don't specify `super`, the JVM assumes `this` by default

`super``(someIdentifier1, someIdentifier2)`

- Substitute the `Class` name of the superclass when you see `super`; therefore, `super` is being used to refer to a constructor in the parent/base/superclass, and the number of parameters passed indicate which constructor is being used in the parent class (assuming there are overloaded constructors present). So when you need to call a superclass constructor *explicitly*, use `super();`

3.5 Using `super` – Notes

1. Subclasses don't inherit constructors; they must call the superclass constructors using `super(…)`

2. Any call to a superclass constructor in a method must appear on the first line of code of a method (just as it did with `this`). You cannot call `super(…)` after any other line of code.

3. As with `this`, you can use `super` to call not just constructors, but methods in the superclass as well. The form is:

```
super.methodName(parameters)
```

3.5 Using `super` – Notes

4. You cannot use `super` to access the private properties of a superclass. For example, using the code seen earlier for the `GeometricObject` superclass and its `Circle` subclass, you may be tempted to try this:

```
public Circle(double radius, String colour, boolean filled){

  setRadius(radius);
  super.colour = colour;
  super.filled = filled;

}
```

`colour` and `filled` are declared private in the superclass `GeometricObject`

…that's what getters and setters are for.

3.5 Using `super` – Constructors – Q & A

Q. You're supposed to provide a no-arg constructor with every class, especially those that will be used as a superclass. Why? What if I don't explicitly use the superclass constructor in my subclass?

A. It doesn't matter. Even if you don't call the superclass constructor, the JVM calls the no-arg superclass constructor for you.

Q. What if I didn't provide a no-arg constructor in the superclass?

A. Then the JVM provides one for you. It silently calls a no-arg constructor in the superclass that it makes *implicitly*. Therefore, as before, it is considered good practice to *always* supply your own no-arg constructor in any class that is going to serve as the parent for a derived class.

3.5 Using `super` – Constructors – Q & A

Q. What if I didn't create a no-arg constructor, but I created other constructors in the superclass, but I don't call them explicitly in the subclass?

A. Then the JVM gets lost. It won't create a no-arg constructor in the superclass since you've already provided constructors. But since you haven't explicitly told it which constructor to use, it flags a compile error. Again, ***always provide a no-arg constructor in the superclass*** to avoid problems.

3.6 Overriding Methods

If you don't like the methods you've inherited from the superclass, you can *override* any superclass method by redefining it in the subclass. This means providing *exactly* the same signature as is found in the superclass.

Returning to our `GeometricObject` example, assume the `GeometricObject` class has the following method:

```
public String toString(){
  return ("GeometricObject created on " + getDateCreated());
}
```

The most important thing about a circle is not the date on which it was created, but the radius of the circle. The `toString()` you've inherited only reports the date. But we can override this method in `Circle()` using:

```
public String toString(){
   return ("This circle object has radius " + getRadius());
}
```

This overrides the superclass's `toString()` method.

3.6 Overriding Methods

So calling on the `toString()` method in an instance of the `Circle` class now calls the 'revised' version of `toString()`:

```
Circle myCircle = new Circle(3.0);
myCircle.toString();
```

This outputs `"This circle object has radius 3.0"`

You can still call on the superclass method using `super`. So

```
super.toString();
```

outputs:

```
GeometricObject created on October 30, 2015
```

Liang 11.4

3.6 Overriding Methods

Another example: if a superclass contains

```
public class MyClass{
    public void message(){
        System.out.println("superClass Rocks!");
    }
}
```

and the subclass contains

```
public class MySubClass extends MyClass{
  public void message(){
     System.out.println(super.message() +
             "but subClass Rules!");
  }
}
```

then if we declare a new instance with `MySubClass msc = new MySubClass`, then the output of `msc.message()` will be

```
superClass Rocks but subClass Rules!
```

3.6 Overriding Methods

Note: Overloading ≠ Overriding

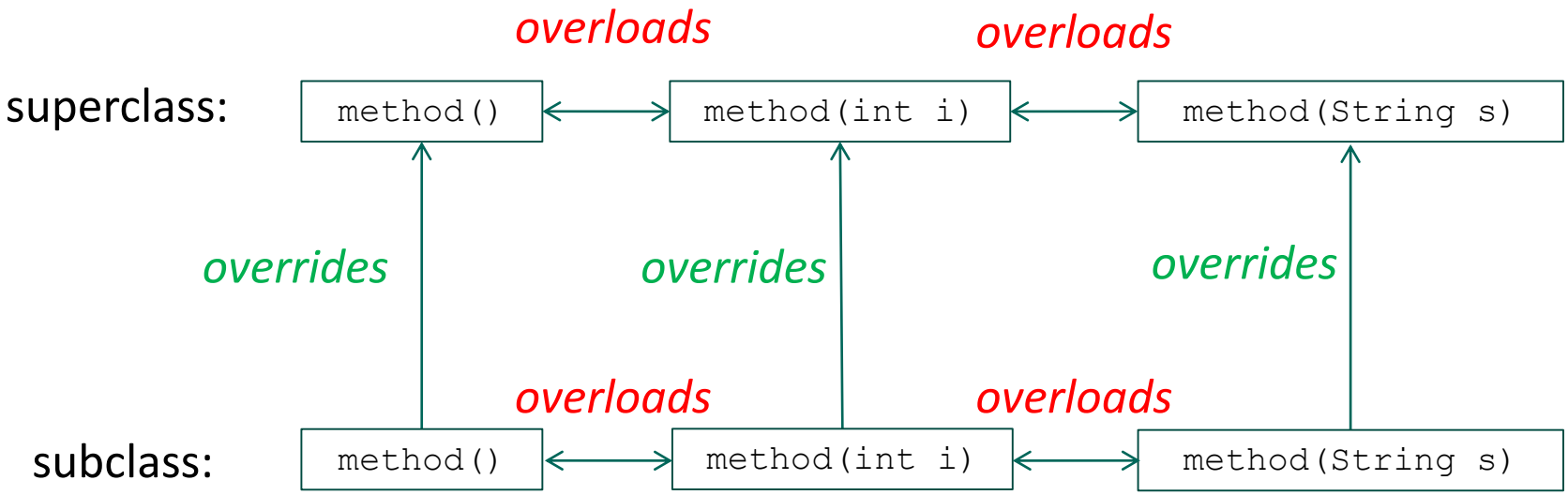
Overload	Override
Uses the same method name, but *different* signature	Uses the same method name and *same* signature
Different code (of course) , but may chain constructors together	Different code, which may call on superclass methods using `super`.
Can overload constructors	Constructors not inherited; they exist only in the parent/base/superclass

Liang 11.5

3.6 Overriding Methods – Q & A

Q. Can I both override and overload methods?

A. Sure, so long as you follow the rules outlined above. Remember: an *overloaded* method has the same name and *different* parameter list; an *overridden* method has the same name and *identical* parameter list to the method in the superclass that it replaces.



3.7 Using @Override

One potential problem of overriding is the possibility of misspelling the method header, or of getting its signature wrong (i.e. different from the superclass). For example what if you meant to write:

```
public String toString(){…}
```

in the subclass, but instead wrote:

```
public String tostring(){…}
```

or possibly

```
public String toStrng(){…}
```

then, since you are adding new methods to the derived class rather than overriding them, Java does not signal any problems: it can't know what you were intending to do.

3.7 Using @Override

Java allows you to catch these potentially time-consuming errors using the `@Override` annotation. Placing this on the line immediately preceding the overridden header declaration causes Java to check that the method exists in the superclass. `@Override` acts as signature-checker.

```
@Override
public String tostring(){…}
```

This gives the error message:

```
"The method tostring() of type Circle must override or
implement a supertype method"
```

thus warning you that `tostring()` does not exist in the superclass.

3.7 Using @Override

Note: *@Override is **NOT** required to override a method found in the superclass. If the subclass signature is exactly the same as the superclass, then you will override correctly, even if you didn't use the @Override annotation.*

Use of @Override is always advisable (and good practice). But its absence is not a mistake…that is, unless you got the subclass signature wrong. Such bugs can be very difficult to detect.

Make sure you understand this.

3.8 The `Object` Class

Every class that is not derived explicitly from another object directly extends the `Object` class. Thus `Object` is 'the mother of all classes'. For example, the declaration of `String` is:

```
public final class String extends Object…
```

and of `Scanner`

```
public final class Scanner extends Object…
```

So whenever you declare a class (including your own) like

```
public class ClassName
```

this is the exact equivalent of

```
public class ClassName extends Object
```

by default.

Every class, including the ones you build, extends directly or indirectly from `Object`.

3.8 The `Object` Class

So what methods and properties are found in `Object`? This is important, since these methods will appear in *every* class (unless you override them!), since every class is ultimately extended from `Object`—including the classes you create.

Object

`+Object()`
 	
`#clone(): Object`
`+equals(obj: Object): boolean`
`+toString(): String`
`#finalize(): void`

D&D 9.6

3.8 The `Object` Class

The methods shown above are mostly self-explanatory. They include such methods as:

- `clone()`, which makes a copy of the object.
- `equals()`, which simply compares two object's reference values and returns `true` if they are equal. The actual implementation of `equals()` is :

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

Liang 11.10

- `finalize()` is triggered by the garbage collector when it determines that there are no more references to the object.

This is a partial list; for the complete list see http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

3.8 The `Object` Class

`toString()` is perhaps the most frequently overridden member of the `Object` class. It provides a default string that can be used to output information on any class to the output device.

In the `Object` class, `toString()` provides (very) basic information about the default object itself. If we don't override `toString()` in the derived class, we can see what this default behaviour looks like. For example, if we call `toString()` from within `Circle` (without overriding it first, as we did in Section 3.6)

```
Circle myCircle = new Circle();
System.out.println(myCircle.toString());
```

we would see output something like:

```
Circle@1F037e5
```

i.e. the class name the object is instantiated from, an @ sign, along with the object's `hashCode*` in hexadecimal.

Liang 11.6

*What is a hashCode? See: http://eclipsesource.com/blogs/2012/09/04/the-3-things-you-should-know-about-hashcode/

3.8 The `Object` Class

In its native form, this is not terribly useful information. In `GeometricObject`, we could override `toString()` to provide somewhat more information, as we did earlier:

```
@Override
public String toString(){
   return "The object " + super.toString() +
    " was created on " + getDateCreated();
}
```

Now when we print out `myCircle.toString()`, we see

```
The object GeometricOjbect@1F037e5 was created on October 17,
2015
```

3.8 The `Object` Class

Note that `toString()` is the default method of any object. Thus you can invoke the `toString()` method simply by using the object's name by itself.

For example

```
System.out.println(GeometricObject)
```

is equivalent to

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

This means whenever you see output like

```
The circle's radius is GeometricObject@1F037e5
```

that's because you tried to print out the object, instead of its intended method, e.g.

```
System.out.println("The circle's radius is " + circle1);
```

3.8 The `Object` Class – Q & A

Q. Can every method in a non-`final` class be overridden?

A. No. Methods, like classes, can themselves be made `final`. This essentially tells the user: you have to use this method as is, you can't override it.

Q. Can static methods be overridden?

A. Well…that's a tricky question. The general answer is 'no', but this doesn't prevent you from creating a new static method with the same name in the subclass. As we noted before, static methods 'live' only once in memory, with the class, and they are not inherited. To override one static method with another would imply that there are two kinds of static methods: a '`super`' method, and an overridden, local '`this`' method. So there would be two static methods with the same name, and this goes against the whole notion of what 'static' is about: it belongs to the class (not the object), and it is loaded only once in memory. The question is somewhat complicated, since it involves compile-time versus run-time issues.

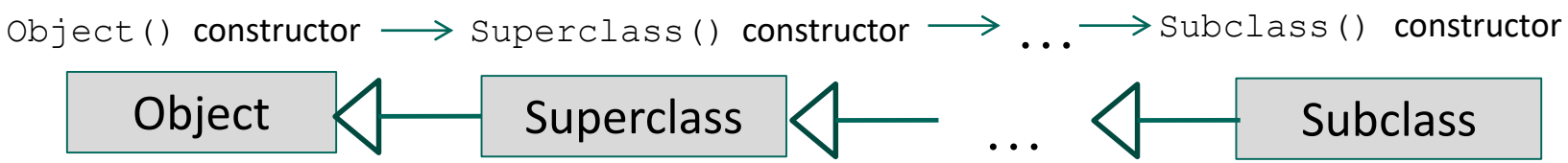
See: http://docs.oracle.com/javase/tutorial/java/IandI/override.html

3.8 The `Object` Class – Q & A

Q. Does the superclass constructor need to be *explicitly* invoked whenever you instantiate an object in the subclass?

A. No. Whenever you create an instance of a subclass, the superclass constructor is invoked *automatically*. Therefore, whenever you instantiate any new object that extends from `Object`, the `Object` constructor is automatically invoked; there's no need to explicitly invoke `super()` in the derived class. Whenever you instantiate from a class that is in turn based on a superclass which itself is based on `Object`, the superclass will again call on *its* superclass constructor (`Object()`) before calling its own constructor, and then turning control over to the subclass constructor.

Thus the order of constructor execution is:



3.9 Using `instanceof`

Suppose we need to distinguish between different types of objects inside a method that takes a `GeometricObject` as its parameter. Inside the method, how does Java know if this object is a `Circle` object, a `Rectangle` object, etc.?

Java contains the `instanceof()` operator to determine the type of an object. For example, assume `displayObjectParameters()` is a static function that displays an object's parameters. Different objects have different parameters, so we need to be able to distinguish between, say, a `Circle` object (with just a `radius`) and a `Rectangle` object (with a `length` and `width`).

The following example shows how the `instanceof` operator can be used to distinguish between objects derived from the same parent class.

Liang 11.9 **D&D 10.5.6**

The following example is derived from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pp. 428-249.

3.9 Using instanceof

```
public class CastingDemo {
   /**Main method */
   public static void main(String[] args) {
      // Create and initialize two new objects
      Object obj1 = new Circle(1);
      Object obj2 = new Rectangle(1, 3);

      // Display the objects' parameters
      displayObjectParams(obj1);
      displayObjectParams(obj2);
}
```

Notice that the two `GeometricObject` objects are passed 'anonymously'; they lose their subclass identity when they are passed as type `Object`.

3.9 Using instanceof

```
public static void displayObjectParams(Object obj){

    // Check for a Circle object
    if (obj instanceof Circle){
       System.out.print("The circle has area " +
          ((Circle)obj).getArea());
       System.out.println(" and radius " +
          ((Circle)obj).getRadius();

    // Check for a Rectangle object
    else if (obj instanceof Rectangle) {
       System.out.print("The rectangle has area " +
          ((Rectangle).obj).getArea());
       System.out.println(" and dimensions " +
          ((Rectangle)obj.getWidth()) + " by " +
          ((Rectangle)obj.getHeight()));
    }
}
```

3.10 Abstract Classes

As stated in Module 1, an *abstract* class is one that can't be instantiated into an object; it must first be extended to a *concrete* class.

Abstract classes contain the common features particular to any category of things. For example, the `GeometricObject` class introduced earlier is a perfect candidate for an abstract class, since it contains the common properties found in all geometric objects. However, there is no such thing as a 'geometric object' *per se*, since that description is far too broad to refer to any *particular kind of geometric object*.

```
public abstract class GeometricObject {
    ...
}
```

Liang 13.2

D&D 10.4

3.10 Abstract Classes

The constructors of an abstract class *can* be `public`, but *should* be made `protected`. Why?

1. Recall that the constructors of any class are designed to return a new instance of the class itself. If a constructor could be used to make a new instance of an abstract class, then this would defeat the purpose of making the class abstract in the first place. So when a class is made abstract, the constructors need to change as well.

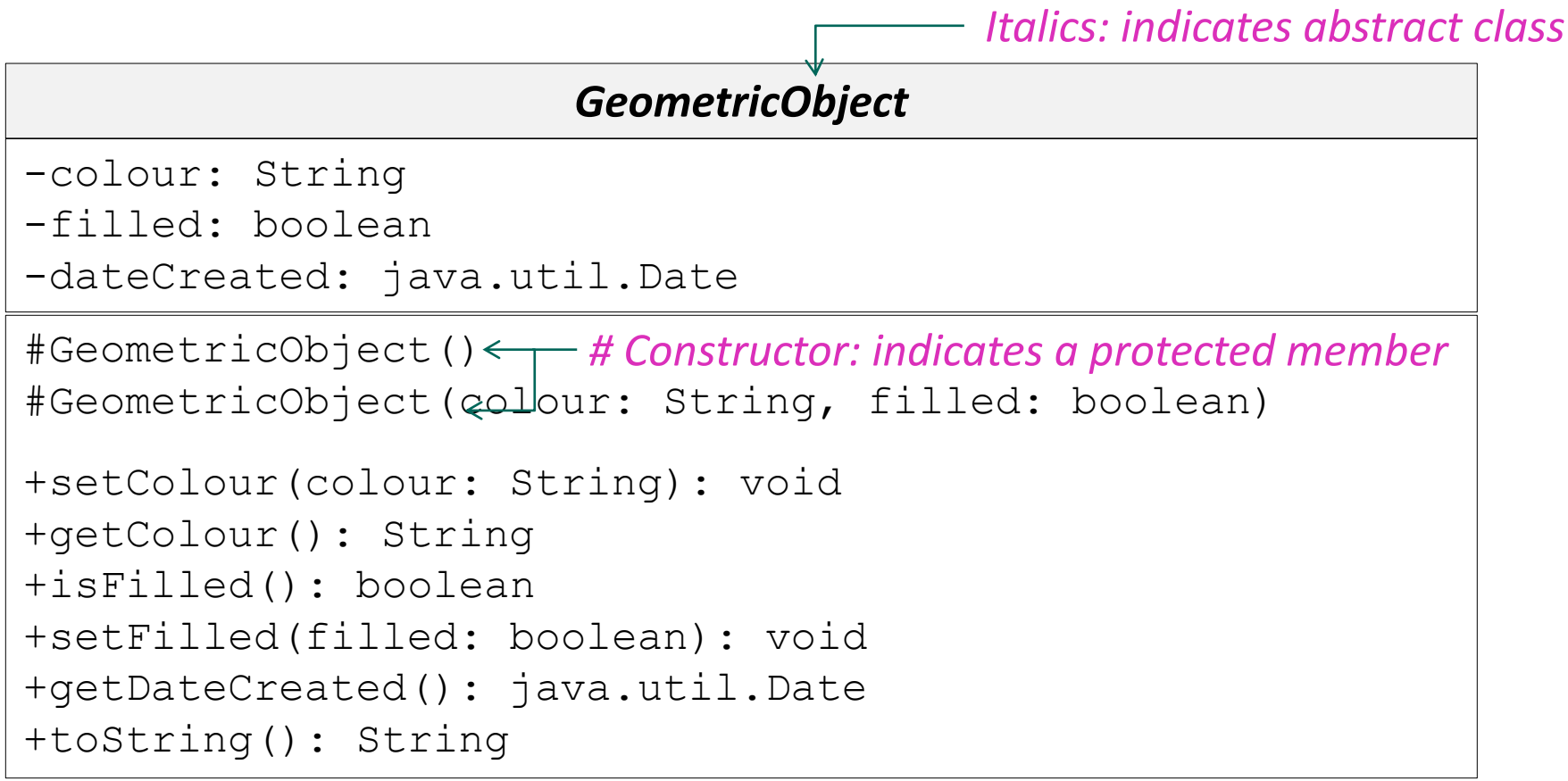
2. You need to be able to *access* an abstract class's constructors (using *super*) in the subclass, but you can't allow the constructor to be used to instantiate a new object of the abstract class directly.

`Protected` fills this requirement. You *can* use protected constructors in the inherited subclass of an abstract superclass , but you *can't* use them to generate new instances of the abstract superclass itself.

This is *not* an essential requirement of an abstract class, but it is considered good practice.

3.10 Abstract Classes

In UML, *italics* are used to indicate that something is abstract. Thus,



3.11 Abstract Methods

Like classes, methods may be made abstract. For example:

```
public abstract class GeometricObject {
    ...
    public abstract double getArea();

}
```

There are two very important things to note about abstract methods:

1. Abstract methods lack a body. There are no curly braces {} after the method header and parameter list, only a semicolon to indicate that the declaration is complete.
2. The word `abstract` must be included in the class header, or the compiler flags an error.

3.11 Abstract Methods

Q. Why would you make a method abstract? What would you use it for?

A. Just as an abstract class may be used to contain a set of fundamental characteristics of some thing (from which other classes may be derived), a method may also embody some fundamental, core feature of a thing, but for which no specific code can be written.

For example, in the `GeometricObject` class, characteristics such as the area and perimeter of a geometric object are obviously important features of that class. But the actual calculation of the area and perimeter differ depending on the actual type of geometric object: a circle has one area, a square another. So abstract methods like `getPerimeter()` and `getArea()` are important features of the `GeometricObject` superclass, but the actual details of their implementation will differ depending on their subclass.

As we'll see later when we deal with interfaces, abstract methods help ensure that any object that extends from an abstract class must have implemented concrete methods that can be used by clients. Thus abstraction forms a kind of enforced contract between an object and its user.

3.11 Abstract Methods

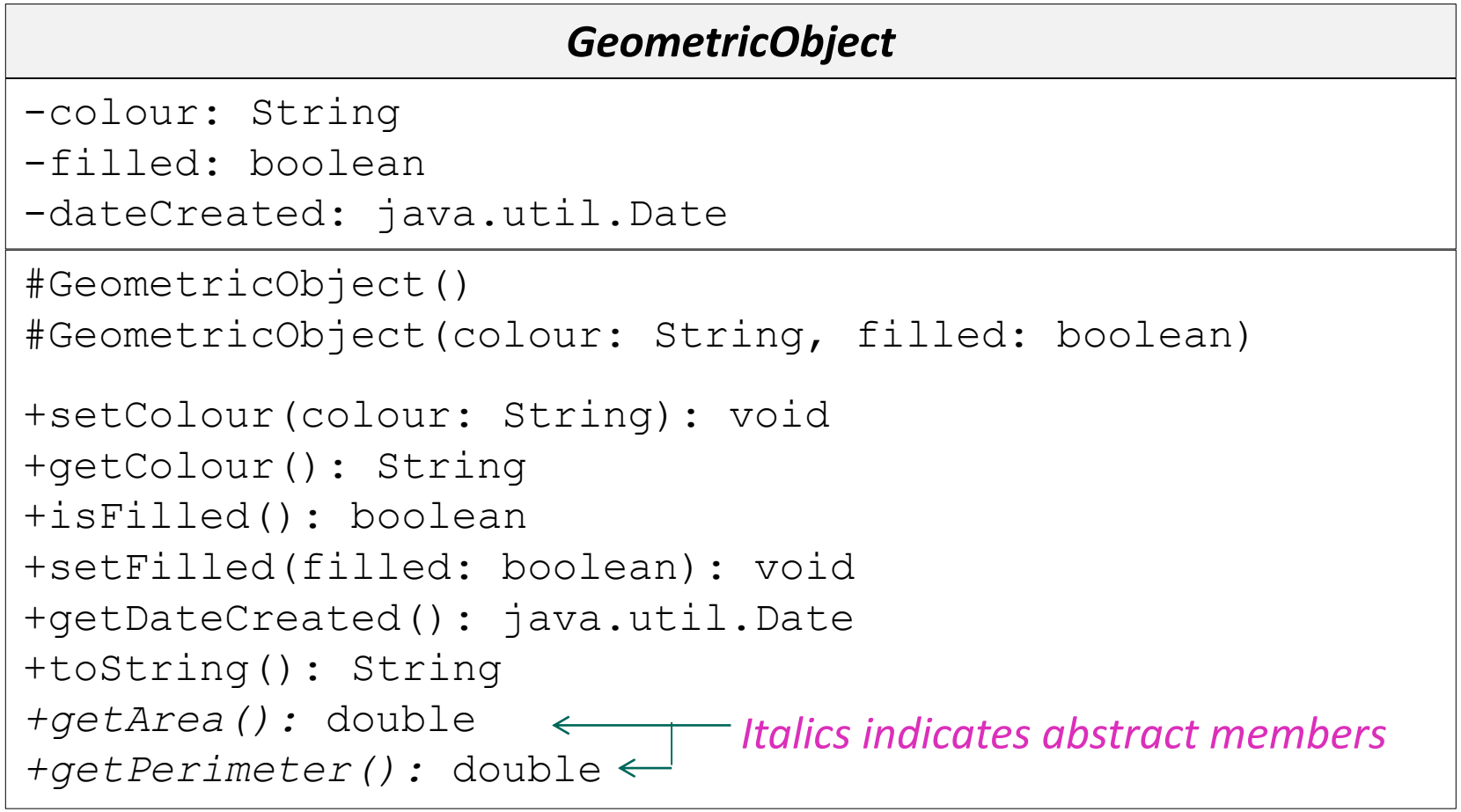
The code corresponding to this UML class diagram looks like this:

```
public abstract class GeometricObject
  private String colour = "white";
  private boolean filled;
  private java.util.Date dateCreated;

  protected GeometricObject(){
    dateCreated =new java.util.Date();
  }

  protected GeometricObject(
        String colour, boolean filled)
    setDate(new java.util.Date());
    setColour(colour);
    setFilled(filled);
  }

  public void setColour(String colour){
    this.colour = colour;
  }

  public String getColour(){
    return colour;
  }

  …etc.

  public abstract double getArea();

  public abstract double getPerimeter();

}
```

abstract methods

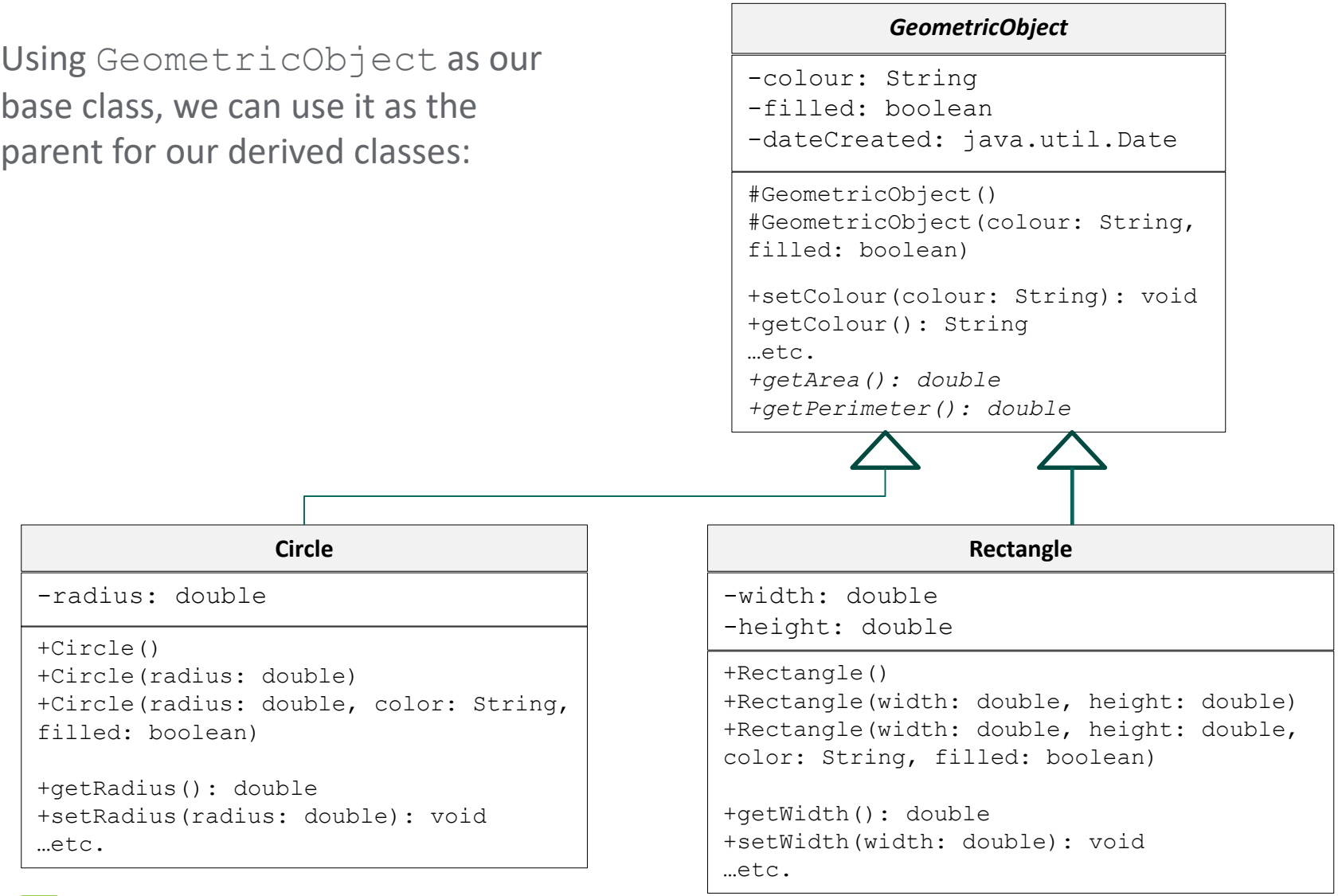
3.11 Abstract Methods

This corresponds to the following UML diagram:



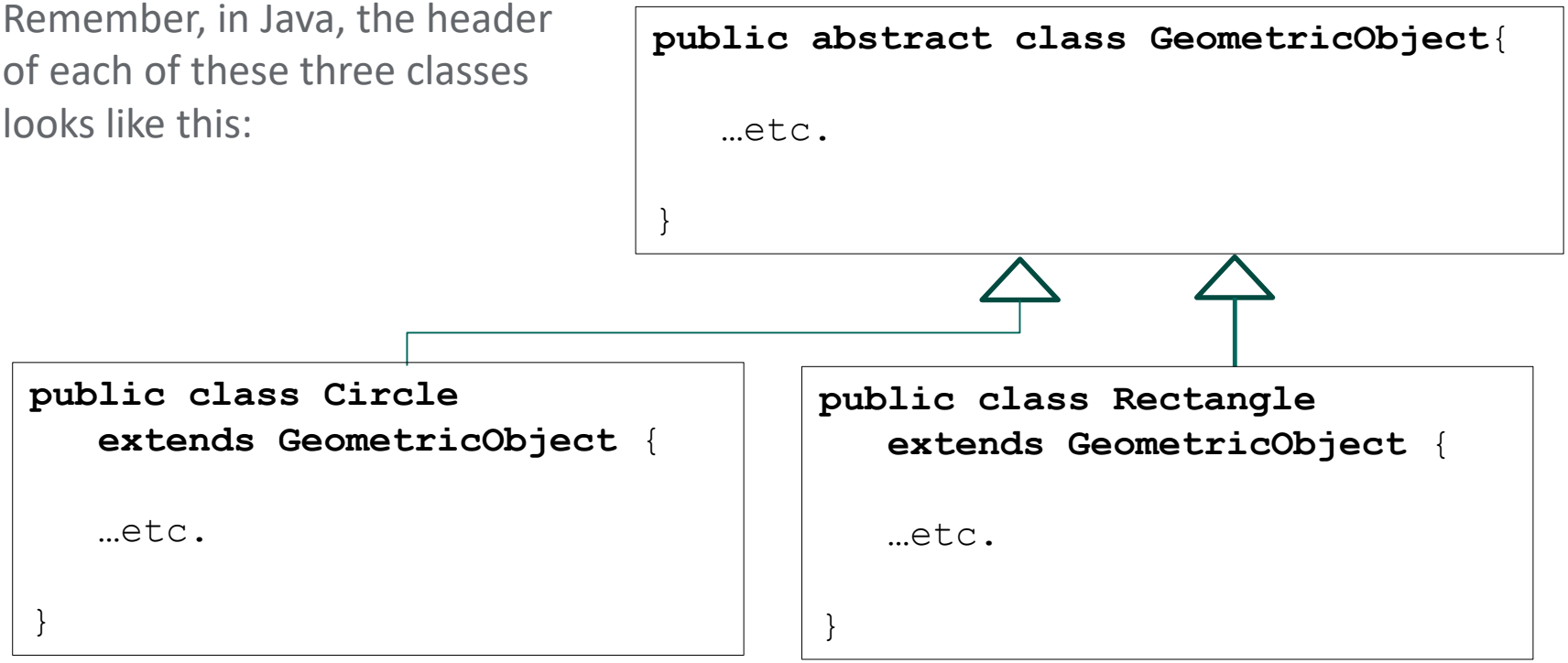
3.11 Abstract Methods

Using `GeometricObject` as our base class, we can use it as the parent for our derived classes:



3.11 Abstract Methods

Remember, in Java, the header of each of these three classes looks like this:



Derived from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pp. 411-415, 496-498.

3.11 Abstract Methods

What about the abstract methods you declared in the `GeometricObject` superclass? They *must* be overridden in the derived class. Therefore, the code in the subclasses will need to contain:

```
public class Rectangle
   extends GeometricObject{
   …
   public double getPerimeter(){
     return (2 * (width + height));
   }

   public double getArea(){
     return (width * height);
   }
}
```

```
public class Circle
   extends GeometricObject{
   …
   public double getPerimeter(){
     return (2 * Math.PI * radius);
   }

   public double getArea(){
     return (Math.PI * radius * radius);
   }
}
```

3.11 Abstract Methods – Notes

1. Whenever a method is declared abstract inside a class, *the whole class itself becomes abstract*. This, of course, does *not* mean that all of its methods become abstract, merely that, as before, you cannot instantiate an object from the class itself.

2. *An abstract method cannot be contained inside a non-abstract class* (based on 1 above), but as we saw earlier, the converse is not true: an abstract class does not need to contain any abstract methods.

3. A subclass does not need to implement all the abstract methods found in the superclass, but only if the subclass is abstract itself. So, for example, `Circle` does not need to override `getArea()` and `getPerimeter()` if `Circle` is declared abstract itself.

4. Despite appearances, `Object` itself is *not* an abstract class.

From: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pp. 500.

3.11 Abstract Methods – Notes

- A subclass can override a superclass method and make it abstract; however, this would be *very* unusual. In this case, the derived class would need to be made abstract (as per the first rule in this list)
- While you cannot create an instance of an abstract class, you can use an abstract class as a data type. For example, given an array of `GeometricObjects`

```
GeometricObject[] geoObjects = new GeometricObject[10];
```

We can load concrete objects of type `Circle` into the above array, which is based on an abstract data type:

```
geoObjects[0] = new Circle();
```

This can then be used to call methods in the abstract class:

```
double circumference = geoObject[0].getPerimeter();
```

So while you can't instantiate an abstract class, that doesn't mean you can't use it in ways associated with instantiated classes.

3.12 Abstraction: The Big Picture

Abstraction is the 'fourth' pillar of OOP. The following definition summarizes the concept rather well:

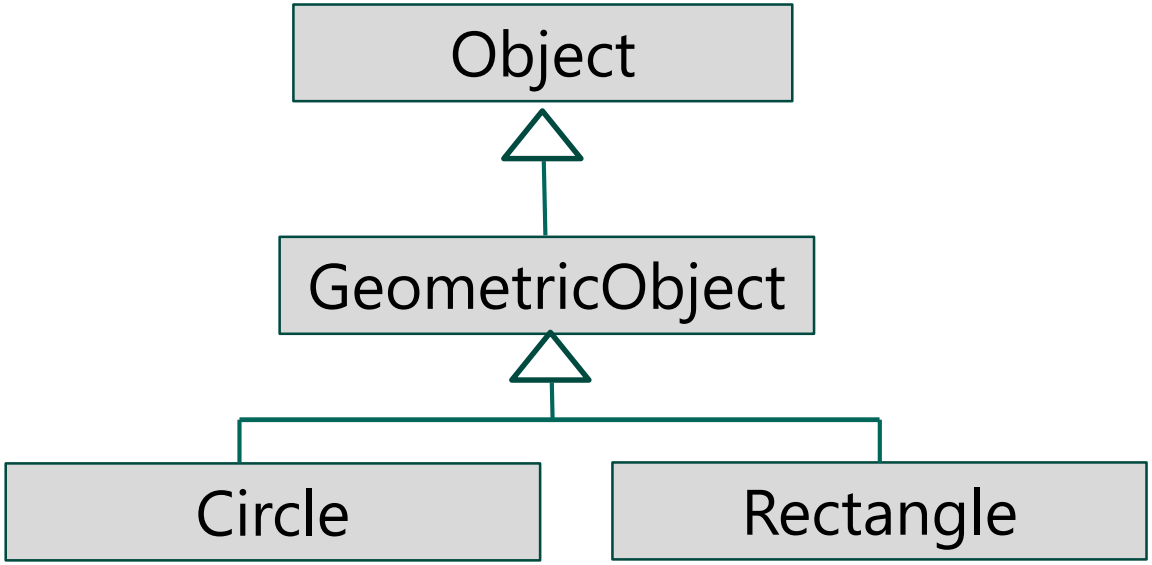
"Abstraction … is the process of taking away or removing characteristics from something in order to reduce it to a set of essential characteristics… Through the process of abstraction, a programmer hides all but the relevant data about an object in order to reduce complexity and increase efficiency. In the same way that abstraction sometimes works in art, the object that remains is a representation of the original, with unwanted detail omitted. The resulting object itself can be referred to as an abstraction, meaning a named entity made up of selected attributes and behavior specific to a particular usage of the originating entity. Abstraction is related to both encapsulation and data hiding."

That's what an abstract class is supposed to do…

From : http://whatis.techtarget.com/definition/abstraction, author unknown, download Sept. 8, 2016

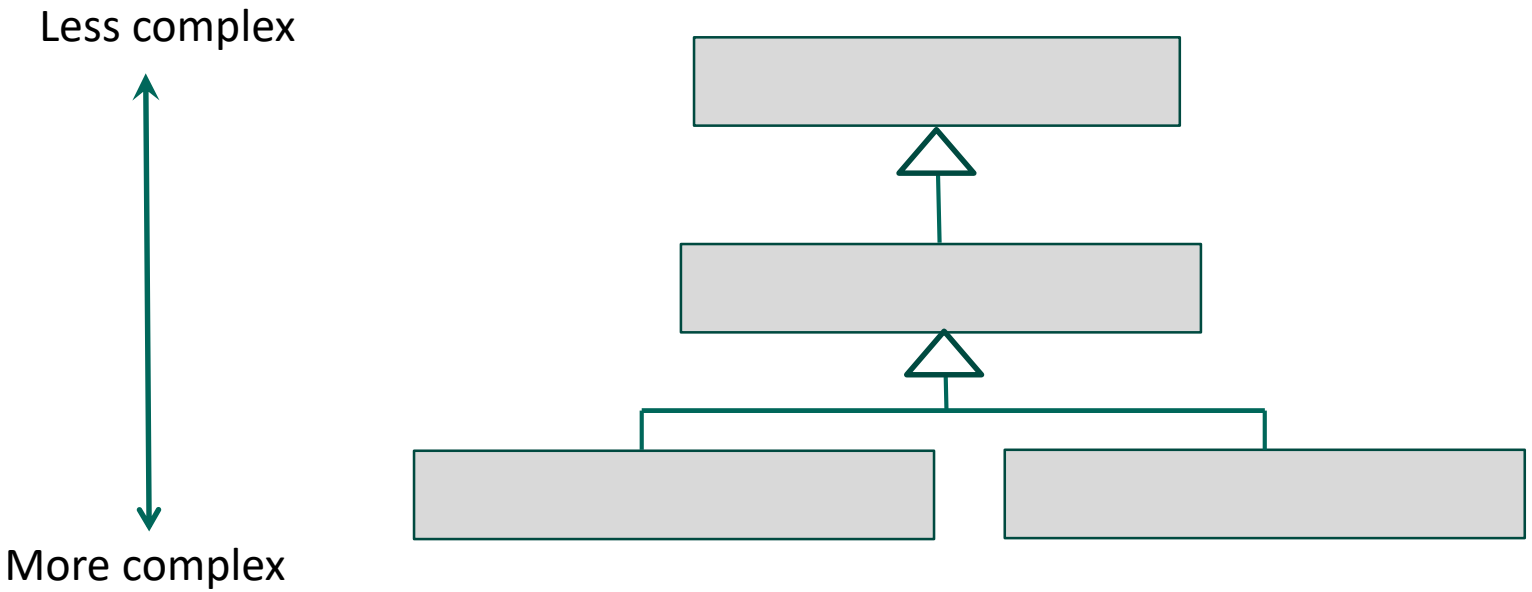
3.12 Abstraction: The Big Picture

Consider the class hierarchy for the objects in this module:



3.12 Abstraction: The Big Picture

Superclasses, like `Object`, contain very few members. As me move *down* the object hierarchy, the number of methods and fields *increases*.



3.12 Abstraction: The Big Picture - Note

The terminology of super- and subclasses can be confusing. Certainly a superclass is better than a subclass? No. *Superclasses are more bas*ic; they can do *less* than subclasses. Thus the members of a *super*class actually form a *sub*set (having fewer items) of the members in the subclass. The *sub*class contains a *super*set (it has more items) of the superclass members. `Super` *does not mean superio*r.

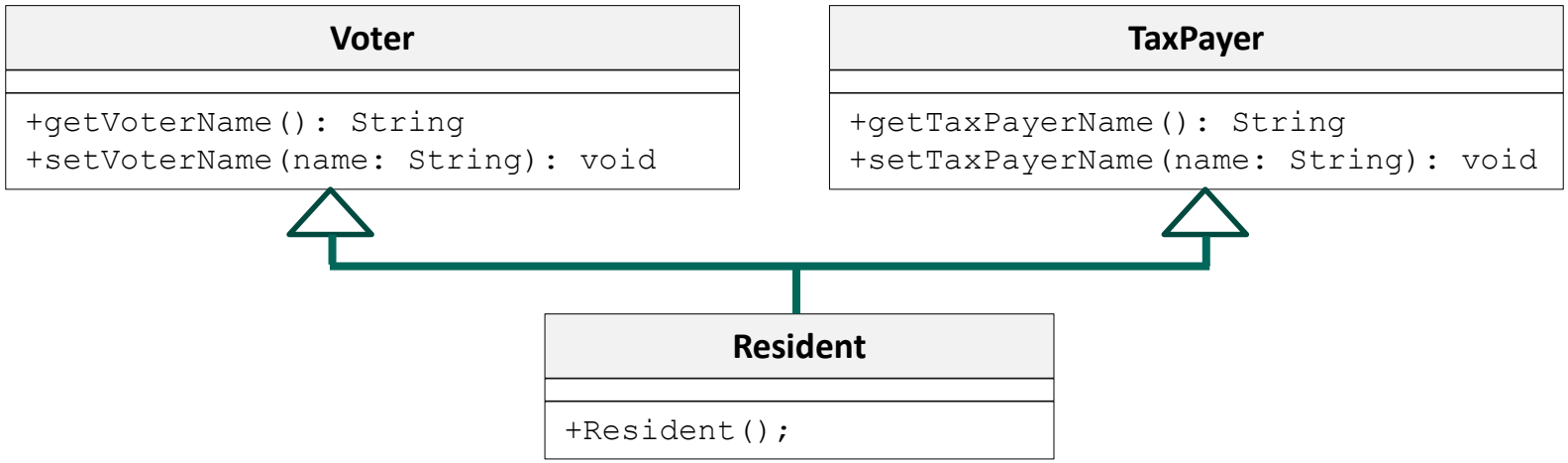
As previously mentioned, a superclass is also referred to as a *base class*. Certainly a base class sits at the bottom of the hierarchy, and *derived classes* are what we are *pointing to* in the UML diagrams? Again, the terminology and the notation can be misleading: base classes are what is being pointed *to*; they sit higher in the hierarchy. Derived classes do the actual pointing in the UML diagram.

Questions

1. You should always use the `@Override` annotation in your subclasses whenever you wish to override a method in the superclass. But in the case of abstract methods in the superclass, you don't explicitly *need* to use the `@Override` annotation for Eclipse to catch the mistake. Why is `@Override` not strictly necessary when overriding abstract methods?

2. What is the difference between a `final` method and an `abstract` method? Can a method be both `final` and `abstract`?

3. Spot four actual or probable errors in the following UML diagram:



Questions

4. Describe how you might accidentally overload an expression when you intended to override it.

5. Assume that two classes, `Square` and `Rectangle`, are to be declared, with one used as the superclass for the other. Given that a subclass always has *as many or more* features that its superclass, which one of the following two statements is more likely to be correct, and explain why:

```
public class Rectangle extends Square {…}
public class Square extends Rectangle {…}
```