Note #3 – Inheritance

- **Inheritance**

Inheritance is an important and powerful feature for reusing software. Suppose you need 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 as to avoid redundancy and make the system easy to comprehend and easy to maintain?

Business are running in a fast pace. Software help the businesses make important decisions and stay competitive. Often times, software must change and adopt new requirements in order to meet the businesses' needs. What would be the best approach to minimize the impact on the running software and incorporate new functionalities into the software?

- **Superclasses (parent classes) and Subclasses (child classes) in Java**

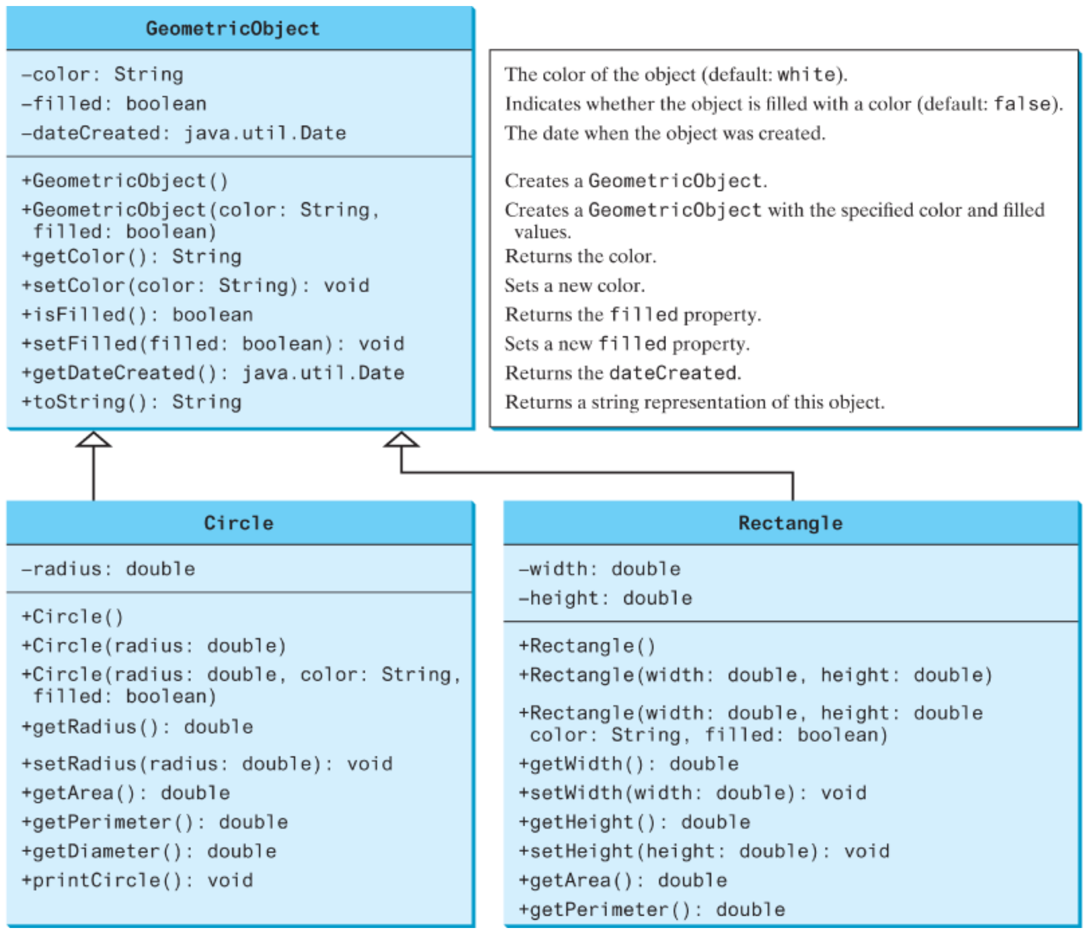
You use a class to model objects of the same type. Different classes may have some **common properties and behaviors**, which can be **generalized** in a class that can be shared by other classes. You can define a specialized class that extends the generalized class. The specialized classes inherit the properties and methods from the general class.

Consider geometric objects. Suppose you want to design the classes to model geometric objects such as circles and rectangles. Geometric objects have many common properties and behaviors. They can be drawn in a certain color and be filled or unfilled. Thus, 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. Assume this class also contains the **dateCreated** property, and the **getDateCreated()** and **toString()** methods. The **toString()** method returns a string representation of the object. Since a circle is a special type of geometric object, it shares common properties and methods with other geometric objects. Thus, it makes sense to define the **Circle** class that extends the **GeometricObject** class. Likewise, **Rectangle** can also be defined as a special type of **GeometricObject**. The following figure shows the relationship among these classes. A triangular arrow pointing to the generalized class is used to denote the inheritance relationship between the two classes involved.

In Java terminology, a class **C1** extended from another class **C2** is called a ***subclass***, and **C2** is called a ***superclass***. A superclass is also referred to as a *parent class* or a *base class*, and a subclass as a *child class*, an *extended class*, or a *derived class*. A subclass inherits accessible data fields and methods from its superclass and may also add new data fields and methods. Therefore, **Circle** and **Rectangle** are subclasses of **GeometricObject**, and **GeometricObject** is the superclass for **Circle** and **Rectangle**. 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***. Therefore, you can say that **Circle** is a subtype of **GeometricObject**, and **-GeometricObject** is a supertype for **Circle**.

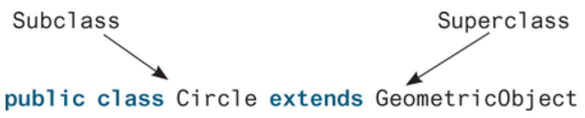
The subclass and its superclass are said to form a *is-a* relationship. A **Circle** object is a special type of general **GeometricObject**. The **Circle** class inherits all accessible data fields and methods from the **GeometricObject** class. In addition, it has a new data field, **radius**, and its associated getter and setter methods. The **Circle** class also contains the **getArea()**, **getPerimeter()**, and **getDiameter()** methods for returning the area, perimeter, and diameter of the circle.

The **Rectangle** class inherits all accessible data fields and methods from the **GeometricObject** class. In addition, it has the data fields **width** and **height** and their associated getter and setter methods. It also contains the **getArea()** and **getPerimeter()** methods for returning the area and perimeter of the rectangle. Note that you may have used the terms width and length to describe the sides of a rectangle in geometry. The common terms used in computer science are width and height, where width refers to the horizontal length, and height to the vertical length.



```
GeometricObject
-color: String                                   The color of the object (default: white).
-filled: boolean                                 Indicates whether the object is filled with a color (default: false).
-dateCreated: java.util.Date                     The date when the object was created.

+GeometricObject()                               Creates a GeometricObject.
+GeometricObject(color: String,                  Creates a GeometricObject with the specified color and filled
 filled: boolean)                                 values.
+getColor(): String                              Returns the color.
+setColor(color: String): void                   Sets a new color.
+isFilled(): boolean                             Returns the filled property.
+setFilled(filled: boolean): void                Sets a new filled property.
+getDateCreated(): java.util.Date                Returns the dateCreated.
+toString(): String                              Returns a string representation of this object.

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 keyword **extends** is used to tell the compiler that the subclass extends the superclass. For example, the **Circle** class above extends the **GeometricObject** class with the following syntax and inherits the methods **getColor**, **setColor**, **isFilled**, **setFilled**, and **toString**.



```
public class Circle extends GeometricObject
```

- **The "super" keyword**

In Java, every class has one and only one direct superclass (single inheritance.) Child class inherits data and operations from the parent, but NOT constructors. Child class can override parent methods it wants to change. Classes that don't explicitly inherit from another class inherit from Object class. You can use a child class wherever a parent class is expected. Some rules a child class must follow. The first line of the constructor must invoke a parent constructor via "super" (superclass). If you override a method and want to get the parent's version, use the **super** key word. For example, **super ()**, or **super(arguments).** If a parent wants to give the children direct access to its data, it must declare it as **protected.**

The statement **super()** invokes the no-arg constructor of its superclass, and the statement **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. Invoking a superclass constructor's name in a subclass causes a syntax error. Using "super" is the only way to explicitly invoke a superclass constructor. For example,

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

A constructor may invoke an overloaded constructor or its superclass constructor. If neither is invoked explicitly, the compiler automatically puts **super()** as the first statement in the constructor. For example:

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

Equivalent

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

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

Equivalent

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

In any case, 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 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. If possible, you should provide a no-arg constructor for every class to make the class easy to extend and to avoid errors.

The keyword **super** can also be used to reference a method other than the constructor in the superclass. The syntax is: **super.method(arguments);**

- **Overriding Methods**

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. **toString()** and **equals()** methods are good examples overriding the behaviors of the superclasses since there are additional data fields in the subclasses; therefore, how you define 2 objects of the subclasses are equal is different.

Note that this is a syntax error: **super.super.methodName();**

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

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

The overriding method must have the same signature as the overridden method and same or compatible return type. Compatible means that the overriding method's return type is a subtype of the overridden method's return type.

An instance method can be overridden only if it is accessible. Thus, 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.

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. The hidden static methods can be invoked using the syntax SuperClassName.staticMethodName.

```java
public class Box {
   protected int x, y;
   //protected means directly accessible by child classes
   protected int width, height;
   public Box( int startX, int startY, int w, int h ) { //constructor
      x = startX;
      y = startY;
      width = w;
      height = h;
   }
   public void show() {
      // Some code that draws the box
   }
   ....
}
```

```
public class ColoredBox extends Box {
   private Color color;
   public ColoredBox (int startX, int startY, int w, int h, Color c ) {
      super( startX, startY, w, h );
      color = c;
   }
   public void show() {
       .....
       super.show();  //if you want to call the show() in the super
class
       .....
   }
   .....
}
```

```
public class Bicycle
{
   protected int cadence;
   protected int gear;
   protected int speed;
   public Bicycle(int startCadence, int startSpeed,
                  int startGear)
   {
      gear = startGear;
      cadence = startCadence;
      speed = startSpeed;
   }
   public void setCadence(int newValue)
   {
      cadence = newValue;
   }
   public void setGear(int newValue)
   {
      gear = newValue;
   }
   public void applyBrake(int decrement)
   {
      speed -= decrement;
   }
   public void speedUp(int increment)
   {
      speed += increment;
   }
}
```

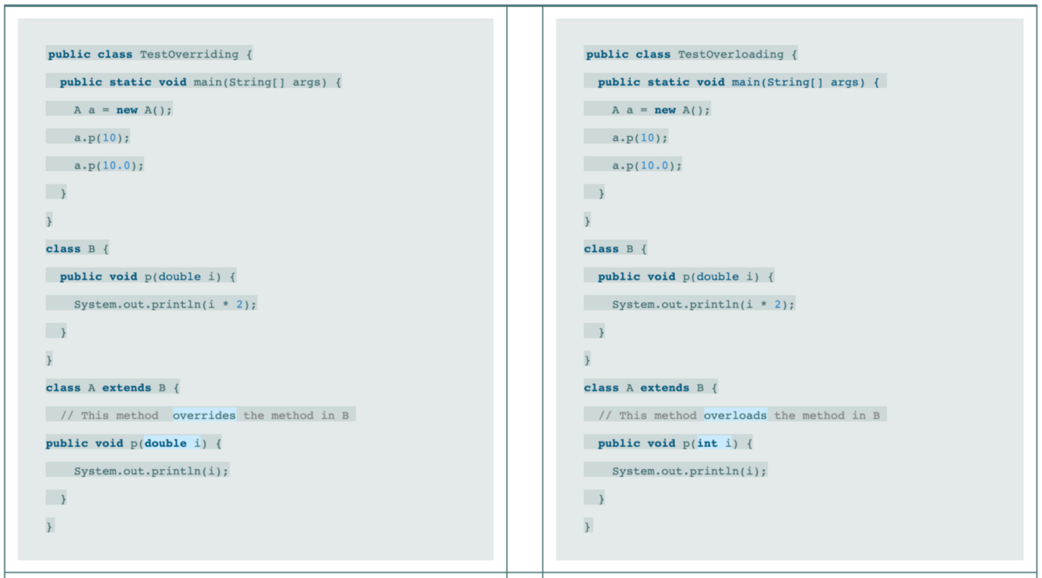
parent class

```
public class MountainBike extends Bicycle
{
   // the MountainBike subclass adds one field
   protected int seatHeight;

   // the MountainBike subclass has one constructor
   public MountainBike(int startHeight,
                       int startCadence,
                       int startSpeed,
                       int startGear)
   {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
   }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue)
    {
       seatHeight = newValue;
    }
}
```

child class

- ## Overriding vs. Overloading

Overloading means to define multiple methods with the same name but different signatures. Overriding means to provide a new implementation for a method in the subclass. For example, in (a) below, the method **p(double i)** in class **A** overrides the same method defined in class **B**. In (b), however, the class **A** has two overloaded methods: **p(double i)** and **p(int i)**. The method **p(double i)** is inherited from **B**.



```
public class TestOverriding {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}
class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}
class A extends B {
  // This method  overrides the method in B
public void p(double i) {
    System.out.println(i);
  }
}
```

(a)

```
public class TestOverloading {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}
class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}
class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

(b)

When you run the **TestOverriding** class in (a), both **a.p(10)** and **a.p(10.0)** invoke the **p(double i)** method defined in class **A** to display **10.0**. When you run the **TestOverloading** class in (b), **a.p(10)** invokes the **p(int i)** method defined in class **A** to display **10** and **a.p(10.0)** invokes the **p(double i)** method defined in class **B** to display **20.0**.

Note that, overridden methods are in different classes related by inheritance; overloaded methods can be either in the same class, or in different classes related by inheritance. Overridden methods have the same signature; overloaded methods have the same name but different parameter lists. To avoid mistakes, you can use a special Java syntax, called ***override annotation***, to place **@Override** before the overriding method in the subclass. This annotation denotes that the annotated method is required to override a method in its superclass. If a method with this annotation does not override its superclass's method, the compiler will report an error. For example, if toString is mistyped as tostring, a compile error is reported. If the @Override annotation isn't used, the compiler won't report an error. Using the @Override annotation avoids mistakes.

- **The Object Class and Its toString() Method**

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 by default. See the Java library class hierarchy here: https://docs.oracle.com/en/java/javase/13/docs/api/overview-tree.html.

Classes such as String, StringBuilder, Loan, and GeometricObject are implicitly subclasses of Object. It is important to be familiar with the methods provided by the Object class so that you can use them in your classes. For example, the toString() method. Invoking toString() on an object returns a string that describes the object. By default, it returns a string consisting of a class name of which the object is an instance, an at sign (@), and the object's memory address in hexadecimal. For example, the output for the following code looks something like Loan@15037e5. This message is not very helpful or informative. Usually you should override the toString method so that it returns a descriptive string representation of the object.

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