

Lecture 3

Learning Objectives

At the end of this lecture, students should

- be familiar with the concept and power of method overriding
- understand how Java dispatches methods that have been overridden
- appreciate the usefulness of overriding `equals` and `toString` in the `Object` class
- be exposed to the `String` class and its associated methods, especially the `+` operator
- be aware that inheritance can be easily abused and leads to bad code
- understand the differences between HAS-A and IS-A relationship
- be able to use composition to model classes with HAS-A relationship
- be able to use inheritance to model classes with IS-A relationship
- understand the Liskov Substitution Principle and thus be aware that not all IS-A relationship should be modeled with inheritance
- understand the purposes of the Java keyword `final`

Java `Object` class

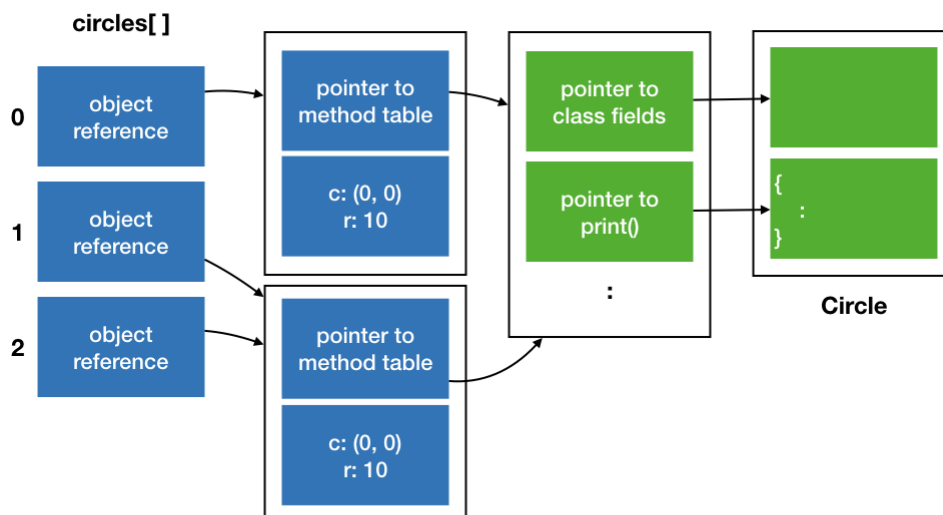
In Java, every class inherits from the class `Object` [<https://docs.oracle.com/javase/9/docs/api/java/lang/Object.html>] implicitly. The `Object` class defines many useful methods that are common to all objects. The two useful ones are :

- `equals(Object obj)` , which checks if two objects are equal to each other, and
- `toString()` , which returns a string representation of the object, and is a better way to print an object than the `print()` method and `Printable` interface we write.

The `equals()` method as implemented in `Object` , only compares if two object references refer to the same object. In the Figure below, we show an array `circles` with three `Circle` objects. All three circles are centered at (0, 0) with radius 10. They are created as follows:

```
1 Circle[] circles = new Circle[3];
2 circles[0] = new Circle(new Point(0, 0), 10);
3 circles[1] = new Circle(new Point(0, 0), 10);
4 circles[2] = circles[1];
```

When you check `circles[0].equals(circles[1])` , however, it returns `false` , because even though `circles[0]` and `circles[1]` are semantically the same, they refer to the two different objects. Calling `circles[1].equals(circles[2])` returns `true` , as they are referring to the same object.



What if you need a method that compares if two circles are *semantically* the same? You can implement your own method, say `isTheSameCircle(Circle c)` . But, the `equals()` method is universal (all classes inherits this method) and is used by other classes for equality tests. So, in most cases, we can implement a method called `equals()` with the same signature with the semantic that we want[8].

That's right. Even though we cannot have two methods with the same signature in the same class, we can have two methods with the same signature, one in the superclass (or the superclass's superclass, and so on), one in the subclass. The method in the subclass will override the method in the superclass. For example,

```
1 class Circle implements Shape, Printable {
2     :
3     @Override
4     public boolean equals(Object obj) {
5         if (this == obj) {
6             return true;
7         }
8     }
```

```

8         if (obj instanceof Circle) {
9             Circle circle = (Circle) obj;
10            return (circle.center.equals(center) && circle.radius == radius);
11        } else {
12            return false;
13        }
14    }
15 }

```

Line 10 above compares if the two center points are equal, and the two radius values are equal. So, we compare if the two circles are semantically the same. The rest of this code requires some explanation:

- Line 3 uses the same `@Override` annotation that we have seen before -- we are telling the compilers that we are overriding a method in the superclass.
- Line 4 declares the method `equals`, and note that it has to have exactly the same signature as the `equals()` method we are overriding. Even though we meant to compare two `Circle` objects, we cannot declare it as `public boolean equals(Circle circle)`, since the signature is different and the compiler would complain.
- Since `obj` is of an `Object` type, we can actually pass in any object to compare with a `Circle`. Line 5 checks if the comparison makes sense, by checking if `obj` is instantiated from a `Circle` class, using the `instanceof` keyword. If `obj` is not even a `Circle` object, then we simply return `false`.
- If `obj` is an instance of `Circle`, we assign `obj` to a variable of type `Circle` and compare as in Line 10.

For the code above to work, we have to override the `equals` method of `Point` as well. That is left as an exercise¹ [fn:1].

One final note: polymorphism works here as well. If we have an object reference `obj` of type `Object` that refers to an instance of a `Circle`, calling `obj.equals()` will invoke the `equals()` method of `Circle`, not `Object`, just like the case of interfaces.

Recall that when a class implements an interface, an instance of that class can take on the type of that interface. Similarly, when a class inherits from a parent class, an instance of that class can take on the type of the parent class. So, we can do the following:

```

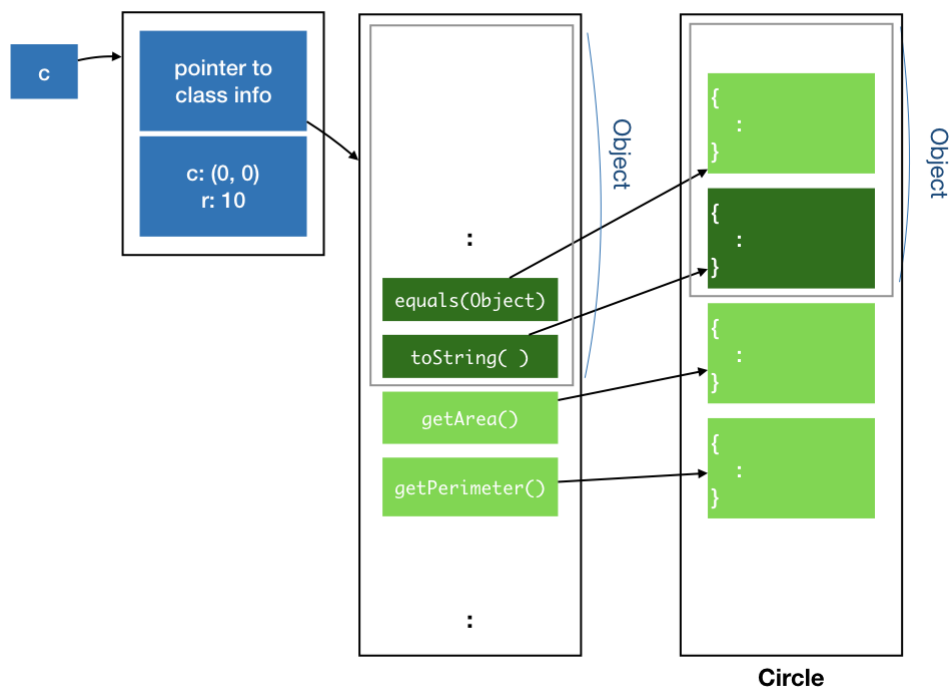
1 Circle c = new Circle(new Point(0,0), 10);
2 Object o = c;

```

Line 2 assigns the circle object `c` to `o` of type `Object`. So, both `o` and `c` are referring to the same objects. Due to type checking at compile time, however, Java does not allow methods and fields in the subclass (e.g., `Circle`) that is not in the superclass (e.g., `Object`) to be called. In this example, only the methods known to `Object` can be accessed by `o`.

Now, consider what would happen if we override the method `equals()` from the `Object` class.

The method table will update the entry for `equals()` to point to the implementation provided by the `Circle` class, instead of the `Object` class.



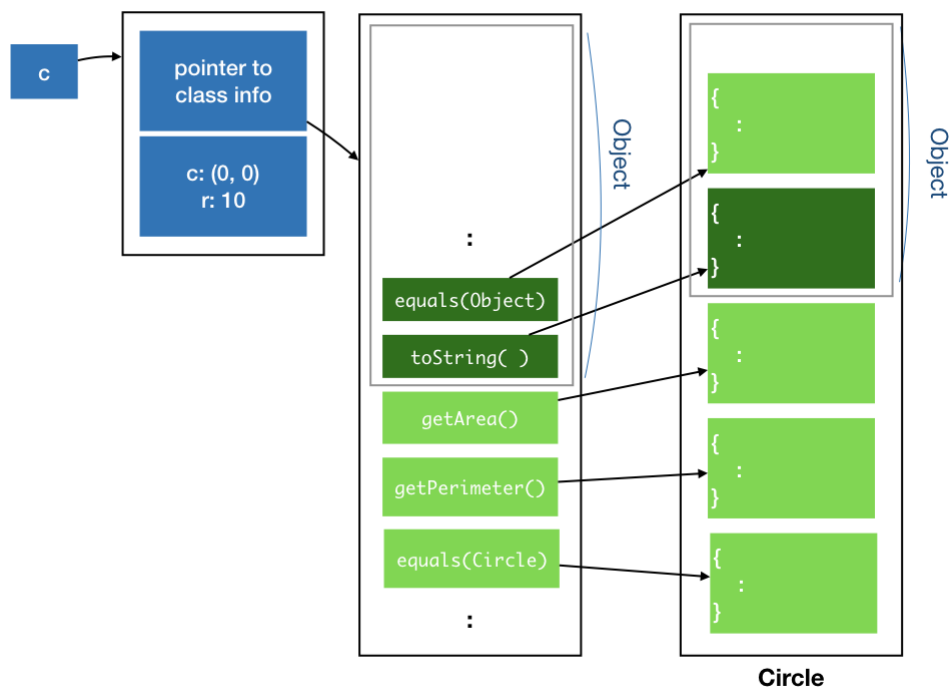
Now, consider what would happen if we *overload* the method `equals()` with one that takes in a `Circle` object. I also throw in a couple of `System.out.print()` to help us figure out what is going on.

```

1 class Circle implements Shape, Printable {
2     :
3     @Override
4     public boolean equals(Object obj) {
5         System.out.println("equals(Object) called");
6         if (obj == this) {
7             return true;
8         }
9         if (obj instanceof Circle) {
10            Circle circle = (Circle) obj;
11            return (circle.center.equals(center) && circle.radius == radius);
12        } else {
13            return false;
14        }
15    }
16
17    public boolean equals(Circle circle) {
18        System.out.println("equals(Circle) called");
19        return ((circle.center.equals(center) && circle.radius == radius);
20    }
21 }

```

Since this new `equals()` method does not override the method in `Object`, it gets its own slot in the method table of `Circle`, instead of reusing one from the `Object`.



Now, consider which version of `equals` are called by the following:

```

1 Circle c1 = new Circle(new Point(0,0), 10);
2 Circle c2 = new Circle(new Point(0,0), 10);
3 Object o1 = c1;
4 Object o2 = c2;
5
6 o1.equals(o2);
7 o1.equals((Circle)o2);
8 o1.equals(c2);
9 c1.equals(o2);
10 c1.equals((Circle)o2);
11 c1.equals(c2);

```

Lines 6-9 call `equals(Object)` defined in `Circle`, while Lines 10-11 call `equals(Circle)` defined in `Circle`. Let's look at why for each one:

- Line 6 calls `equals` of an `Object` object on an `Object` object. Java checks through all methods defined in `Object`, and finds a method that matches the signature, which is the `equals(Object)` (which `Circle` has overridden). This is the one that will get called.
- Line 7-8 call `equals` of an `Object` object on a `Circle` object. Java checks through all methods defined in `Object` and finds one method that matches the signature, which is `equals(Object)` (which `Circle` has overridden). Even though there is another method `equals(Circle)` defined, it is defined as part of the `Circle` class, which Java refuses to access because `o1` is declared to have the type `Object`. Since the only method that Java can find has an argument of type `Object`, the argument is cast as an `Object` when `equals` is invoked.
- Line 9 calls `equals` of a `Circle` object on an `Object` object. Java finds a method with the matching signature, `equals(Object)`, and invokes that.
- Finally, Lines 10-11 call `equals` of a `Circle` object on a `Circle` object. Even though there are two overloaded methods which Java can call without type error, Java always invokes *the most specific method*, in this case, `equals(Circle)`.

Why do we need to override `equals` in `Object`, rather than just using the `Circle`-specific `equals(Circle)`? As shown above, only when an object declared as `Circle` calls `equals` on another `Circle` object, the `Circle`-specific `equals(Circle)` is invoked.

To write code that is general and reusable, we should exploit OO polymorphism, that means different subclasses of `Object` implement their own customized version of `equals`, and the right version of `equals` will be called.

One example of where this is called the `contains(Object)` method from class `ArrayList` (we will cover this later in class), which checks if an object is already in the list, and to check this, it checks for equality of the given object with every object in the `ArrayList`, by calling `equals(Object)`.

toString

We now turn our attention to another method in `Object` that we could override, the `toString()` method. `toString()` is called whenever the `String` representation of an object is needed. For instance, when we try to print an object. By default, the `toString` of `Object` simply prints the name of the class, followed by `@`, followed by the reference. It is often useful to override this to include the content of the object as well, for debugging and logging purposes. This is a much more useful and flexible way than writing our own `print()` method as we have seen in earlier lectures, since we are not limited to printing to standard output anymore.

java.lang.String
String is one of the many useful classes provided by Java. You can skim through to see what methods are available and keep the [API reference](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) [https://docs.oracle.com/javase/8/docs/api/java/lang/String.html] handy.

```

1 class Point {
2     :
3     public String toString() {
4         return "(" + x + ", " + y + ")";
5     }
6 }

```

Now, if we run:

```



1 Point p = new Point(0,0);
2 System.out.println(p);

```

It should print `(0,0)` instead of `Point@1235de`.


The ability to override methods that you inherit from a parent, including root class `Object`, makes overriding an extremely powerful tool. It allows you to change how existing libraries behave, and customize them to your classes, without changing a single line of their code or even accessing their code!

As Uncle Ben said, "With great power, comes great responsibility." We must use overriding and inheritance carefully. Since we can affect how existing libraries behave, we can easily break existing code and introduce bugs. Since you may not have access to these existing code, it is often tricky to trace and debug.

 **Using `super` To Access Overridden Methods** 

After a subclass overrides a method in the superclass, the methods have been overridden can still be called, with `super` keyword. For instance, the following `toString` implementation of `Point` calls the `toString` from `Object`, to prefix the string representation of `Point` with the class and reference address.


```
1 @Override
2 public String toString() {
3     return super.toString() + " (" + x + ", " + y + ")";
4 }
```



The `protected` and Default Access Modifiers

In the last lecture, when we inherit `Circle` from `PaintedShape`, we set the fields `fillColor` etc to `private`, to create an abstraction barrier between the superclass and its subclasses. This barrier allows the implementor of the superclass to freely change the internal representation of the superclass without worrying about the effect on the subclasses.

Sometimes, the implementor of a superclass may choose to allow the subclasses to have access to some of its fields and methods, but yet prevent other classes from accessing them. This type of access restriction can be achieved with the `protected` access modifier.

 **`protected` in Other Languages**

C++ and C# both provide `protected` keyword, allowing subclasses to access `protected` fields and methods of the superclass. Swift, however, decided that [deciding fields/methods access based on inheritance complicates things without bringing any advantage](https://developer.apple.com/swift/blog/?id=11) [https://developer.apple.com/swift/blog/?id=11] and does not provide the `protected` access modifier.

Java Packages

So far we have written several classes and interfaces (`Circle`, `Point`, `Shape`, `Printable`, `PaintedShape`, `Square`, etc). We are using common names to name our classes and interfaces, and it is not inconceivable that in a large software project using external libraries that we will end up with multiple classes with the same name! For instance, Java library provides a `Point` [https://docs.oracle.com/javase/9/docs/api/java/awt/Point.html] class and a `Shape` [https://docs.oracle.com/javase/7/docs/api/java/awt/Shape.html] interface as well.

Java `package` mechanism allows us to group relevant classes and interfaces together under a *namespace*. You have seen two packages so far: `java.awt` where we import the `Color` class from, and `java.lang` where we import the `Math` class from. These are provided by Java as standard libraries. We can also create our own package and put the classes and interfaces into the same package. We (and the clients) can then import and use the classes and interfaces that we provide.

Besides providing namespace to disambiguate classes or interfaces with the same name, Java `package` also provides another higher-layer of abstraction barrier. In Java, a `protected` field or method can be accessed by other classes in the same package.

Finally, Java has the forth access modifier known as the default modifier. This access modifier (or lack of it) is used when we do not specify `public`, `protected`, nor `private`. A field or member with no access modifier is private to the package -- it is `public` to all classes within the same package, but `private` to classes outside of the package. The default access modifier is also known as *package-private* by some.

The following table, taken from [Oracle's Java Tutorial](https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html) [https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html] summarizes the access modifiers:

Access Modifier	Class	Package	Subclass	World
<code>public</code>	Y	Y	Y	Y
<code>protected</code>	Y	Y	Y	N
no modifier	Y	Y	N	N
<code>private</code>	Y	N	N	N

In Java, every class belongs to a package, whether we like it or not. If we do not declare that a class belongs to a package, then it belongs to the default package.


We will not be discussing `package` much more than what we have done today. You can learn more about packages via [Oracle's Java Tutorial](https://docs.oracle.com/javase/tutorial/java/package/index.html) [https://docs.oracle.com/javase/tutorial/java/package/index.html].

Modeling HAS-A Relationship

Inheritance in OO tends to get overused. *In practice, we seldom use inheritance.* Let's look at some examples of how *not* to use inheritance, and why.

You may come across examples online or in books that look like the following:

```
1 class Point {
2     protected double x;
3     protected double y;
4     :
5 }
6
7 class Circle extends Point {
8     protected double radius;
9     :
10 }
11
12 class Cylinder extends Circle {
13     protected double height;
14     :
15 }
```



`Circle` implemented like the above would have the center coordinate inherited from the parent (so it has three fields, `x`, `y`, and `radius`), and `Cylinder` would have the fields corresponding to a circle, which is its base, and the height. So, we are *reusing* the fields and the code related to initializing and manipulating the fields.

When we start to consider methods encapsulated with each object, things start to get less intuitive. What does `getPerimeter()` and `getArea()` of `Cylinder` means? How about `distanceTo` between a `Cylinder` and a `Point`? What is the meaning of a `Circle` containing a `Cylinder`?

The inheritance hierarchy above actually models the HAS-A relationship: A circle has a center (which is a point), a cylinder has a base which is a circle. Therefore, a better way to capture the relationship between the three types of objects is through *composition*:

```
1 class Point {
2     double x;
3     double y;
4     :
5 }
6
7 class Circle {
8     Point center;
9     double radius;
10    :
11 }
12
13 class Cylinder {
14     Circle base;
15     double height;
16     :
17 }
```

Composition allows us to build more complex classes from simpler ones, and is usually favored over inheritance.

The `PaintedShape` class from Lecture 2, for instance, could be modeled as a composition of a `Style` object and `Shape` object.

```
1 class Style {
2     Color fillColor;
3     Color borderColor;
4     :
5 }
6
7 class PaintedShape {
8     Style style;
9     Shape shape;
10    :
11    public double getArea() {
12        return shape.getArea();
13    }
14    :
15    public void fillWith(Color c) {
16        style.fillWith(c);
17    }
18    :
19 }
```

The design above is also known as the *forwarding*-- calls to methods on `PaintedShape` gets forwarded to either `Style` or `Shape` objects.

Modeling IS-A Relationship

A better situation to use inheritance is to model a IS-A relationship: when the subclass behaves just like parent class, but has some additional behaviors. For instance, it is natural to model a `PaintedCircle` as a subclass of `Circle` -- since a `PaintedCircle` has all the behavior of `Circle`, but has *additional* behaviors related to being painted.

```
1 class PaintedCircle extends Circle {
2     Style style;
3     :
4 }
```

A more tricky situation for modeling a IS-A relationship occurs when the subclass behaves just like the parent class *most* of the time, but sometimes behave slightly differently than the parent. Consider how we model a rectangle and a square. Normally, we consider a square IS-A special case of a rectangle. So, we could model as:

```
1 class Rectangle {
2     double width, height;
3     Point topLeft;
4     Rectangle(Point topLeft, in width, int height) {
5         this.topLeft = topLeft;
6         this.width = width;
7         this.height = height;
8     }
9 }
10
11 class Square extends Rectangle {
12     Square(Point topLeft, int width) {
13         super(topLeft, width, width);
14     }
15 }
```

So far, so good.

Now, suppose the two classes are written by two different developers. The developer who wrote `Rectangle` decided to add the method `resizeTo`:

```
1 class Rectangle {
2     :
3     void setSize(int width, int height) {
4         this.width = width;
5         this.height = height;
6     }
7 }
```

This developer assumes the behavior that, after calling `setSize(w, h)`, the width of the rectangle will be `w` and the height will be `h`. He/she publishes this API, and another developer then assumes this behavior, and wrote some code, like:

```
1 void doSomething(Rectangle r) {
2     r.setSize(1, 2);
3     :
4 }
```

What should the developer who develops `Square` do? Since `Square` is a subclass of `Rectangle`, it would inherit `setSize` from its parent, but it does not make sense to call `setSize` with two different parameters. Sure, `Square` can overload `setSize` and provide a `setSize` with one parameter only, but that does not prevent someone from calling `setSize` with two parameters on a `Square`. Someone could do the following and the code would still compile and run, turning the square into a rectangle!

```
1 Square s = new Square(new Point(0,0), 10);
2 s.setSize(4, 8);
```

The `Square` developer can try to override `setSize`, to ignore the second parameter:

```
1 class Square extends Rectangle {
2     :
3     @Override
4     void setSize(int width, int height) {
5         this.width = width;
6         this.height = width;
7     }
8 }
```

This makes more sense and would make everyone who uses `Square` happy -- a square is always a square -- but it introduces an *inconsistency* in behavior and will most likely break another part of the code that the developer is totally unaware of. The developer of `doSomething` suddenly cannot assume that `setSize` works as intended and documented.

It is a developer's responsibility that any inheritance with method overriding does not alter the behavior of existing code. This brings us to the *Liskov Substitution Principle* (LSP), which says that: "Let $\phi(x)$ be a property provable about objects x of type T . Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T ."

This means that if S is a subclass of T , then an object of type T can be replaced by an object of type S without changing the desirable property of the program.

In the example above, this means that everywhere we can expect rectangles to be used, we can replace a rectangle with a square. This was no longer true with the introduction of `setSize` method.

Preventing Inheritance and Method Overriding

Sometimes, it is useful for a developer to explicitly prevent a class to be inherited. Not allowing inheritance would make it much easier to argue for the correctness of programs, something that is important when it comes to writing secure programs. Both the two java classes you have seen, `java.lang.Math` and `java.lang.String`, cannot be inherited from. In Java, we use the keyword `final` when declaring a class to tell Java that we ban this class from being inherited.

```
1 final class Circle {
2     :
3 }
```

Alternatively, we can allow inheritance, but still prevent a specific method from being overridden, by declaring a method as `final`. Usually, we do this on methods that are critical for the correctness of the class.

```
1 class Circle {
2     :
3     final public boolean contains(Point p) {
4         :
5     }
6 }
```

final variable

The keyword `final` has another use. When declaring a variable as `final`, just like `PI` in `Math`, it prevents the variable from being modified. In other words, the variable becomes constant.

```
1 public static final double PI = 3.141592653589793;
```

Exercise

- In the class `Point`, add a new method `equals` that overrides the `equals` from `Object`, so that when calling `p.equals(q)` on two `Point` objects, the method return `true` if and only `p` and `q` coincide (i.e., have the same coordinates).
- Consider the following classes: `FormattedText` adds formatting information to the text. We call `toggleUnderline()` to add or remove underlines from the text. A `URL` is a `FormattedText` that is always underlined.

```
1 class FormattedText {
2     public String text;
3     public boolean isUnderlined;
4     public void toggleUnderline() {
5         isUnderlined = !isUnderlined;
6     }
7 }
8
9 class URL extends FormattedText {
10    public URL() {
11        isUnderlined = true;
12    }
13    public void toggleUnderline() {
14        // do nothing
15    }
16 }
```

Does it violate the Liskov Substitution Principle? Explain.

- Consider each of the code snippets below. Will it result in a compilation or run time error? If not, what will be printed?

(a)

```
1 class A {
2     void f() {
3         System.out.println("A f");
4     }
5 }
6
7 class B extends A {
8 }
9
10 B b = new B();
11 b.f();
12 A a = b;
13 a.f();
```

(b)

```
1 class A {
2     void f() {
3         System.out.println("A f");
4     }
5 }
```

```

6
7 class B extends A {
8     void f() {
9         System.out.println("B f");
10    }
11 }
12
13 B b = new B();
14 b.f();
15 A a = b;
16 a.f();
17 a = new A();
18 a.f();

```

(c)

```

1 class A {
2     void f() {
3         System.out.println("A f");
4     }
5 }
6
7 class B extends A {
8     void f() {
9         super.f();
10        System.out.println("B f");
11    }
12 }
13
14 B b = new B();
15 b.f();
16 A a = b;
17 a.f();

```

(d)

```

1 class A {
2     void f() {
3         System.out.println("A f");
4     }
5 }
6
7 class B extends A {
8     void f() {
9         this.f();
10        System.out.println("B f");
11    }
12 }
13
14 B b = new B();
15 b.f();
16 A a = b;
17 a.f();

```

(e)

```

1 class A {
2     void f() {
3         System.out.println("A f");
4     }
5 }
6
7 class B extends A {
8     int f() {
9         System.out.println("B f");
10        return 0;
11    }
12 }
13
14 B b = new B();
15 b.f();
16 A a = b;
17 a.f();

```

(f)

```

1 class A {
2     void f() {
3         System.out.println("A f");
4     }
5 }
6
7 class B extends A {
8     void f(int x) {
9         System.out.println("B f");
10        // return x; <-- this line should not be here.
11    }
12 }
13
14 B b = new B();
15 b.f();
16 b.f(0);
17 A a = b;
18 a.f();
19 a.f(0);

```

(g)

```

1 class A {
2     public void f() {
3         System.out.println("A f");
4     }
5 }
6
7 class B extends A {
8     public void f() {
9         System.out.println("B f");
10    }
11 }
12
13 B b = new B();
14 A a = b;

```

```
15 a.f();
16 b.f();
```

(h)

```
1 class A {
2     private void f() {
3         System.out.println("A f");
4     }
5 }
6
7 class B extends A {
8     public void f() {
9         System.out.println("B f");
10    }
11 }
12
13 B b = new B();
14 A a = b;
15 a.f();
16 b.f();
```

(i)

```
1 class A {
2     static void f() {
3         System.out.println("A f");
4     }
5 }
6
7 class B extends A {
8     public void f() {
9         System.out.println("B f");
10    }
11 }
12
13 B b = new B();
14 A a = b;
15 a.f();
16 b.f();
```

(j)

```
1 class A {
2     static void f() {
3         System.out.println("A f");
4     }
5 }
6
7 class B extends A {
8     static void f() {
9         System.out.println("B f");
10    }
11 }
12
13 B b = new B();
14 A a = b;
15 A.f();
16 B.f();
17 a.f();
18 b.f();
```

(k)

```
1 class A {
2     private int x = 0;
3 }
4
5 class B extends A {
6     public void f() {
7         System.out.println(x);
8     }
9 }
10
11 B b = new B();
12 b.f();
```

(l)

```
1 class A {
2     private int x = 0;
3 }
4
5 class B extends A {
6     public void f() {
7         System.out.println(super.x);
8     }
9 }
10
11 B b = new B();
12 b.f();
```

(m)

```
1 class A {
2     protected int x = 0;
3 }
4
5 class B extends A {
6     public void f() {
7         System.out.println(x);
8     }
9 }
10
11 B b = new B();
12 b.f();
```

(n)


```

1  class A {
2      protected int x = 0;
3  }
4
5  class B extends A {
6      public int x = 1;
7      public void f() {
8          System.out.println(x);
9      }
10 }
11
12 B b = new B();
13 b.f();

```

(o)

```

1  class A {
2      protected int x = 0;
3  }
4
5  class B extends A {
6      public int x = 1;
7      public void f() {
8          System.out.println(super.x);
9      }
10 }
11
12 B b = new B();
13 b.f();

```

4. Consider each of the code snippets below. Which will result in a compilation error?

(a)

```

1  class A {
2      public void f(int x) {}
3      public void f(boolean y) {}
4  }

```

(b)

```

1  class A {
2      public void f(int x) {}
3      public void f(int y) {}
4  }

```

(c)

```

1  class A {
2      private void f(int x) {}
3      public void f(int y) {}
4  }

```

(d)

```

1  class A {
2      public int f(int x) {
3          return x;
4      }
5      public void f(int y) {}
6  }

```

(e)

```

1  class A {
2      public void f(int x, String s) {}
3      public void f(String s, int y) {}
4  }

```

1. If you override `equals()` you should generally override `hashCode()` as well, but let's leave that for another lesson on another day.