The `static` Keyword and Inheritance

Today, we look at when and how to use the `static` keyword in Java. We then examine ***inheritance***, one of the pillars of object-oriented design.

Manager: Recorder:

Presenter: Reflector:

Content Learning Targets

After completing this activity, you should be able to say:

- I can differentiate between static and instance variables and methods.
- I can summarize best practices for when to use static variables and methods.
- I can explain what it means for one class to extend another and summarize the `extends` and `super` keywords.
- I can generalize multiple classes that have overlapping code.
- I can explain the requirements of abstract classes and methods.
- I can write a new method for an existing Java library class.

Process Skill Goals

During the activity, you should make progress toward:

- Reading Java API documentation and making inferences. (Information Processing)
- Making conclusions based on IDE hints and program output. (Critical Thinking)

Model 1 The `static` Keyword

We have seen the `static` Java keyword come up in a few situations. It's time for a deep dive into what it means and how to use it.

Version 1:

```
public class StudentV1 {
    private String name;
    private char grade;

    public StudentV1(String name, char grd) {
        this.name = name;
        this.grade = grd;
    }

    @Override
    public String toString() {
        return name + " earned " + grade;
    }

    public static void main(String[] args) {
        StudentV1 a =
            new StudentV1("Adaline", 'A');
        StudentV1 b =
            new StudentV1("Belicia", 'B');
        StudentV1 c =
            new StudentV1("Charlie", 'C');
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
    }
}
```

Version 2:

```
public class StudentV2 {
    private String name;
    private static char grade;

    public StudentV2(String name, char grd) {
        this.name = name;
        StudentV2.grade = grd;
    }

    @Override
    public String toString() {
        return name + " earned " + grade;
    }

    public static void main(String[] args) {
        StudentV2 a =
            new StudentV2("Adaline", 'A');
        StudentV2 b =
            new StudentV2("Belicia", 'B');
        StudentV2 c =
            new StudentV2("Charlie", 'C');
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
    }
}
```

Questions (20 min) Start time:

1. Examine the two versions of a simple `Student` class above. Notice the subtle differences, and predict the output of each. ***After*** predicting, run them (in `src/student`) to confirm.

`StudentV1` output:

`StudentV2` output:

2. The English word ***static*** can take several different meanings. Based on this example, which meaning do you think Java is using?

3. In the Java API docs for the `Integer` class, find the list of fields. What are some of the `static` fields in this class? Click on each field's name to see more info.

4. Look at `static` fields in other Java classes, such as `Math`, `Calendar`, and `HttpURLConnection`. Based on what you find, what do you think are some patterns/conventions/best practices for using `static` fields?

Methods can also be declared as `static`, meaning they are associated with the class itself and not with individual instances. In fact, we can call static methods (a.k.a. class methods) without ever creating an instance of the object type. A static method has ***unchanging*** behavior: it is not state-dependent, meaning it does not do different things depending on the data stored in the current object instance. Static methods are (almost always) true functions, in the mathematical sense: for each valid input, they will give the same output every time.

5. We have already used one static method in the `Integer` class: `Integer.parseInt(String s)`. (Notice the camel-case naming convention and the `ClassName.methodName()` syntax.) Look at the list of `static` methods in `Integer`. What are some typical use cases for static methods?

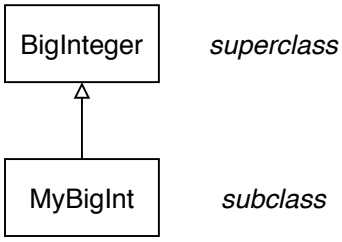
6. We always declare the `main` method in Java as `static`. Why do you think this is necessary?

7. Examine the provided ***Point.java*** file. Uncomment the print statements, and add the missing methods. Which one is static? How do the two distance methods differ in "point" of view?

Model 2 Extending Classes: My Big Integer

The following class extends the functionality of `BigInteger` to allow comma-separated strings (e.g., `"123,465,789"`). The UML diagram summarizes the relationship between the two classes.

```
1  import java.math.BigInteger;
2
3  public class MyBigInt extends BigInteger {
4
5      public MyBigInt(String val) {
6          // remove comma characters
7          super(val.replace(",",  ""));
8      }
9
10     public String toString() {
11         // start with the decimal representation
12         String str = super.toString();
13         StringBuilder sb = new StringBuilder(str);
14
15         // insert comma separators every three digits
16         for (int i = sb.length() - 3; i > 0; i -= 3) {
17             sb.insert(i, ',');
18         }
19         return sb.toString();
20     }
21
22 }
```



Questions (20 min) Start time:

8. Based on the UML diagram, how do we indicate an `extends` relationship in UML?

9. The keyword `super` behaves like the keyword `this`, except that it refers to the superclass. On the following lines, which method (in which class) is being invoked?

a) Line 7:

b) Line 11:

c) Line 18:

10. Open *MyBigInt.java*. Copy the following code snippets into the `main` method, one at a time (without the others), and run them. Record the results in the table below.

Java Code	Result
`BigInteger bi = new BigInteger("123456789");` `System.out.println(bi);`	
`MyBigInt bi = new MyBigInt("123456789");` `System.out.println(bi);`	
`BigInteger bi = new BigInteger("123,456,789");` `System.out.println(bi);`	
`MyBigInt bi = new MyBigInt("123,456,789");` `System.out.println(bi);`	

`BigInteger bi1 = new BigInteger("123456789");`
`MyBigInt bi2 = new MyBigInt("123,456,789");`
`System.out.println(bi1.equals(bi2));`
`System.out.println(bi2.equals(bi1));`	

11. Based on these results, summarize what the source code for each method does:

a) `MyBigInt` constructor

b) `MyBigInt.toString`

c) `MyBigInt.equals`

12. Why do you think `bi2.equals(bi1)` compiles and runs correctly, even though the `MyBigInt` class does not define an `equals` method?

13. Refer to the documentation for BigInteger and the source code for `MyBigInt`. How many public items are defined in each class?

a) `BigInteger` fields:

b) `BigInteger` constructors:

c) `BigInteger` methods:

d) `MyBigInt` fields:

e) `MyBigInt` constructors:

f) `MyBigInt` methods:

14. Type the code on the right in `main` and view possible completions suggested by the IDE.

a) How many constructors does a `MyBigInt` have?

b) About how many methods does a `MyBigInt` have?
(not counting the `main` method)

```
bi2 = new MyBigInt(

bi2.
```

15. Notice that `MyBigInt` has most of the same fields and methods as `BigInteger`. Non-private fields and methods are ***inherited*** when extending a class. Based on your answers to the previous two questions, what is not inherited? Explain your reasoning.

16. Make the following changes to *MyBigInt.java*, and summarize the compiler errors.

a) Rewrite the constructor using two lines of code:

```
String str = val.replace(",", "");
super(str);
```

b) Remove all code from the body of the constructor.

c) Remove the constructor altogether.

17. Consider a method `isPalindrome()` that determines whether a `MyBigInt` has the same digits forward and backward. For example, `123,321` and `12,321` are palindromes, but `123,421` and `12,341` are not. How could you implement this method?

```
public boolean isPalindrome() {




}
```

18. Add your solution to *MyBigInt.java*, and make sure it works. What code can you add to `main` to test the `isPalindrome` method?

Model 3 Abstract Classes

Just like in language and philosophy there are abstract ideas and categories that can be realized as concrete examples/things, Java allows us to distinguish between `abstract` classes and ***concrete*** (the default) classes.

```
public class ToySheep {
    private int volume;

    public ToySheep() {
        this.volume = 3;
    }

    public int getVolume() {
        return volume;
    }

    public void setVolume(int volume) {
        this.volume = volume;
        makeNoise();
    }

    public void makeNoise() {
        System.out.println("Baaa");
    }
}
```

```
public class ToyRobot {
    private int chargeLevel;
    private int volume;

    public ToyRobot() {
        this.chargeLevel = 5;
        this.volume = 10;
    }

    public void recharge() {
        chargeLevel = 10;
    }

    public int getVolume() {
        return volume;
    }

    public void setVolume(int volume) {
        this.volume = volume;
        makeNoise();
    }

    public void makeNoise() {
        System.out.println("Beep Beep!");
    }
}
```

Questions (25 min) Start time:

19. Identify ***similarities*** in the code: what fields and methods do the classes have in common?

20. Summarize ***differences*** between the two classes.

21. Design a new class named `LoudToy` that contains the code that `ToySheep` and `ToyRobots` have in common. Its constructor should take `volume` as a parameter, and `makeNoise` should have an empty body.

```
public class LoudToy {









}
```

22. Redesign `ToySheep` so that it extends `LoudToy`. The constructor of `ToySheep` should call the constructor of `LoudToy`. Remove the code from `ToySheep` that is no longer necessary.

```
public class ToySheep extends LoudToy {









}
```

23. Redesign `ToyRobot` so that it extends `LoudToy`, and remove extraneous code.

```
public class ToyRobot extends LoudToy {




}
```

24. What is the output of the following examples?

a)
```
LoudToy toy1 = new LoudToy(1);
toy1.makeNoise();
```

b)
```
LoudToy toy2 = new ToySheep();
toy2.makeNoise();
```

c)
```
LoudToy toy3 = new ToyRobot();
toy3.makeNoise();
```

Notice that the ***instantiated type*** of an object can be a subclass of its variable's ***declared type***. In other words, we can store `ObjectType` in a variable with declared type `DeclaredType` if and only if `ObjectType` "is-a" `DeclaredType`.

25. In #24, did the variable's ***declared*** type or the object's ***instantiated*** type determine the version of `makeNoise` that was called?

26. Would it ever make sense to construct a `LoudToy` object? Why/why not?

The `abstract` keyword can be used to declare methods that have no body, forcing subclasses to override them. Classes with abstract methods must also be defined as abstract.

```
public abstract class LoudToy {
    private int volume;

    public LoudToy(int volume) {
        this.volume = volume;
    }

    public int getVolume() {
        return volume;
    }

    public void setVolume(int volume) {
        this.volume = volume;
        makeNoise();
    }

    public abstract void makeNoise();
}
```

27. Summarize the differences between Model 3 and your answer to #21.

28. Open *LoudToy.java* (from Model 3) in your IDE. Remove the word `abstract` from the class definition. What are the two compiler errors?

29. Replace the word `abstract` in the class definition, and then remove the word `abstract` from the method definition. What is the compiler error now?

30. Remove the definition of `makeNoise` altogether, and notice the compiler error. Why is it necessary to declare this method in `LoudToy`?

31. Undo all changes in *LoudToy.java*, and add the following `main` method. What is the compiler error message? Why do you think Java doesn't allow you to construct a `LoudToy`?

```
public static void main(String[] args) {
    LoudToy toy1 = new LoudToy(1);
    toy1.makeNoise();
}
```

32. Open *ToySheep.java* and rename `makeNoise` to `makeNoise2`. What is the compiler error?

33. Rename the method back to `makeNoise`, but change `void` to `int`. What is the error now?

34. Explain how an abstract method is like a contract.