Lecture - Classes and Methods - II

Acknowledgement

It has been transferred from static slides to Ed Stem by Lachlan Andrew, with minor changes.

Review

- Class definitions
  - Class structure
  - Variables
  - Methods
- Encapsulation
  - Access modifiers (e.g., public vs private)
  - Accessor and mutator methods
- Overloading
- Constructors

Outline

- Static methods and static variables
    - The Math class and wrapper classes
    - Automatic boxing and unboxing mechanism
- References and class parameters
    - Variables and Memory
    - Using and misusing references
- Packages
- Javadoc

Static methods

A *static method* is one that can be used without a calling object

A static method still belongs to a class, and its definition is given inside the class definition

When a static method is defined, the keyword static is placed in the method header

```
public static returnedType myMethod(parameters)
{ . . . }
```

Static methods are invoked using the class name in place of a calling object.

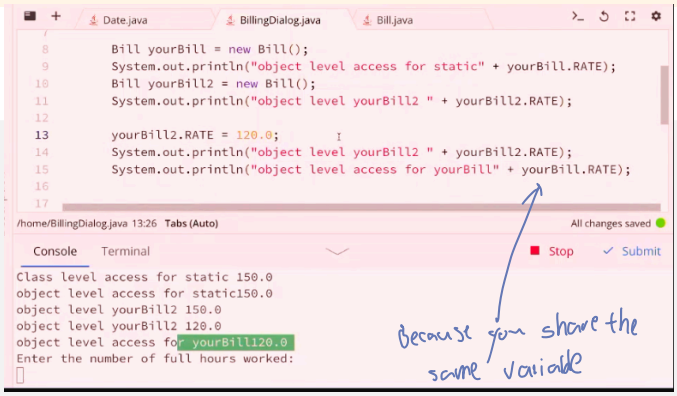
```
returnedValue = MyClass.myMethod(arguments);
```

⚠ Static methods cannot call non-static methods

What is wrong with the following code?

```
class Main {
    private void sayHello () {
        System.out.println ("Hello, World!");
    }

    public static void main (String[] args) {
        sayHello ();
    }
}
```



Main main = new Main();

⇒ not work in this case

or using private static void sayHello()

When a *non-static* method is called, it is passed a hidden parameter, `this`, which refers to the object ("instance") of this class from which it is being called. A static method can be called without an object, like `Main.main()`, and so there is no `this` variable that it can pass.

Exercise: Fix the above code in one or both of the following ways:

1. Make `sayHello()` static too.
2. In `main()`, create a new variable called `instance` of class `Main`, and then call `instance.sayHello()`

Exercise: Why does each of these work?

... Main in any class

Although the `main` method is often by itself in a class separate from the other classes of a program, it can also be contained within a regular class definition.

There can be *multiple* classes with a `public static void main (String[] args)` method.

The system knows which one to start with by the way the compiler is called. We call `java classname` and it is the `main()` in `classname.java` that is called when the program starts.

Why have more than one `main()`? If there is one class per file, it can contain diagnostic code or a self-test, so that the class can be compiled by itself as a single unit to test any changes made to the class.

```
class Main {
    public static void main (String[] args) {
        System.out.println("Hello, world!");

        Other.main(new String[0]);
    }
}

class Other {
    public static void main (String[] args) {
        System.out.println("Goodbye.");
    }
}
```

:) Hello, World!
Goodbye.

See also Display 5.3 of the text book.

Static variables

A static variable is a variable that belongs to the class, and not to one object of that class.

- There is only one copy of a static variable per class.
- There is a separate copy of an instance variable for each object of the class

static variable belongs to class

instance variable belongs to objects.

All objects of the class can read and modify a static variable

Although a static method cannot access an instance variable, it can access a static variable.

Exercise: Why?

A static variable is declared like an instance variable, with the addition of the modifier `static`

```
private static int myStaticVariable;
```

...Initialization

Static variables can be declared and initialized at the same time

if not initialise, they'll have default value.

```
private static int myStaticVariable = 0;
```

If not explicitly initialized, a static variable will be automatically initialized to a default value

- `boolean` static variables are initialized to `false`
- Other primitive types static variables are initialized to the zero of their type
- Class-type static variables are initialized to `null`

It is always preferable to initialize static variables explicitly rather than rely on the default initialization

...Example: Human

The following example shows a simple use of a static variable and a static method to access that method.

It creates multiple objects of class Human, and keeps track of the number of such objects there are.

```
public class Main {
    public static void main (String[] args) {
        for (int i = 0; i < 10; i++) {
            Human newHuman = new Human ("Person " + i);
            System.out.println("Current population: "
                + Human.getPopulation());
        }
    }
}

class Human {
    private String name;
    private static int populationCount = 0;

    public Human(String aName) {
        name = aName;
        populationCount++;
    }

    public static int getPopulation() {
        return populationCount;
    }
}
```

Advanced: This is actually slightly misleading, because the counter is incremented when an object is created, but not decremented when it is destroyed. The actual objects are only referred to temporarily by the `newHuman` variable. In the next iteration of the loop, the java virtual machine is allowed to destroy the previous `Human` object, since nothing refers to it any more. That means the actual population may be less than the value shown.

If you want to, you can create a `protected void finalize()` function. If a `Human` object is removed (in a process called garbage collection) then this function will be called. It can reduce `populationCount`, and print the new population. You will probably find that this is never called, unless you add `System.gc();` to the end of `main()`. In that case, I found that only one object was garbage collected, but your mileage may vary.

Constants never gonna change

Static variables should usually be defined `private`.

This is in contrast to *constants*. These are declared like variables, but with the `final` modifier, which means that their values cannot be changed. This makes it safe to make them `public`.

```
public static final double PI = 3.14159;
```

When referring to one of these outside its class, use the name of its class in place of a calling object

```
int myVar = RoundStuff.PI;
```

Share price

uppercase for the name.
upper snake case

This exercise is to create a class `Holding` that represents a single investment on the share market. A `Holding` object records how many shares a person holds in a particular stock.

The class Holding also records the current value of each share. This is common to all `Holding` objects.

Use your knowledge of static methods and static variables to create this class.

A test class SharePrice has been written, which indicates what methods are required. It should be called from the console (`>_`) as

```
$ java  SharePrice  sharesInAccount1  sharesInAccout2  pricePerShare
```

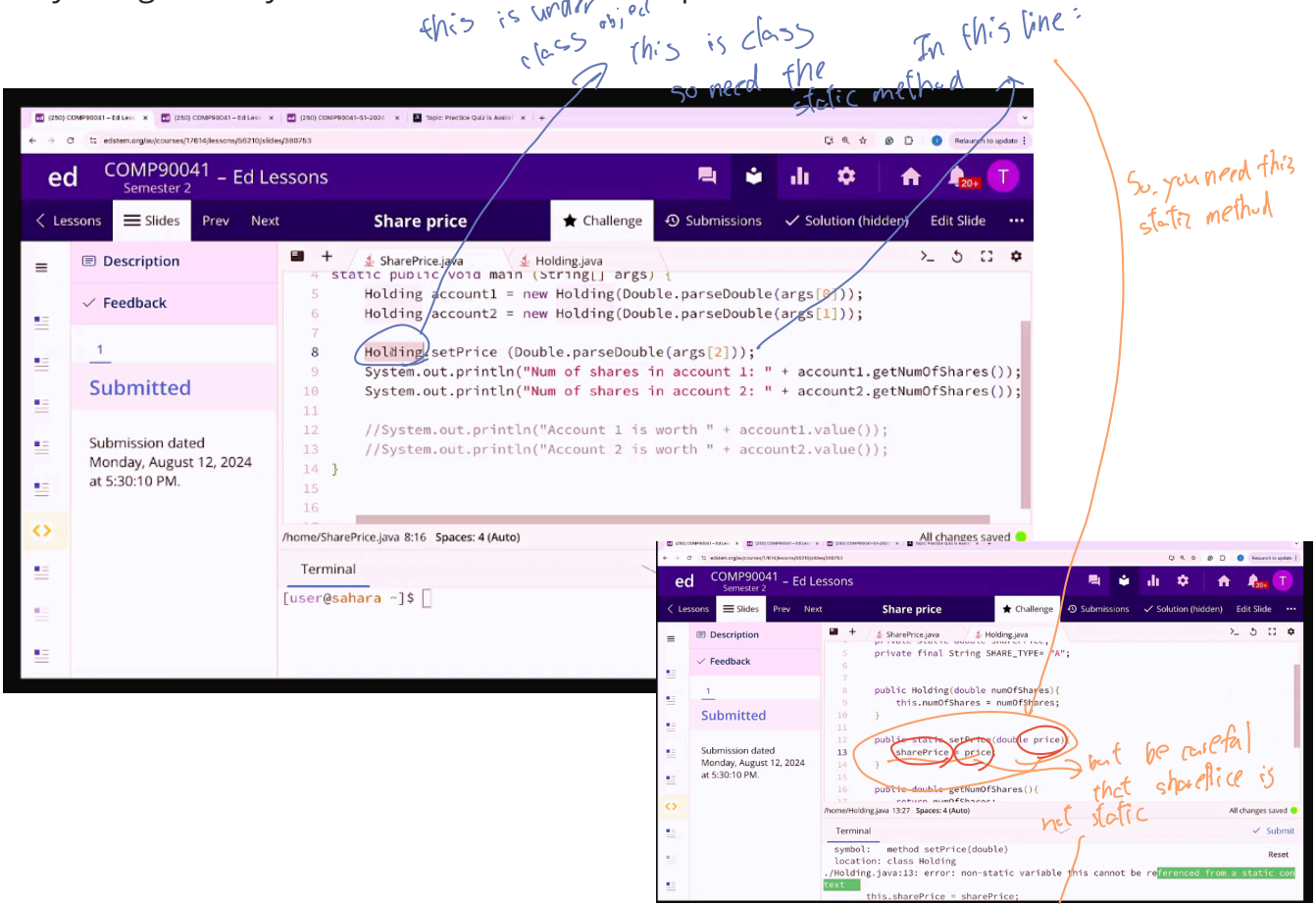
e.g.

```
$ java   SharePrice  11   87   43.6
```

There are several static methods defined or used in the scaffold. Which are they? How can you tell they are static?

Extension

You can extend the `SharePrice` and `Holding` classes to allow a dividend (cents per share) to be paid, and reinvested by using it to buy more shares at the current price.



Nested, Inner and Anonymous Classes

Nested and Inner Classes

The Java language allows you to define a class within a class. Such a class is called a **Nested Class**. Nested classes are divided into two categories: non static and static. Non static classes are called **inner classes**. The classes that are defined as static are called static nested classes.

Here is an example code that demonstrates the usage of both inner and static nested classes

```
public class OuterClass {
    private int outerData = 10;

    // Static Nested Class
    public static class NestedClass {
        public void display() {
            System.out.println("Nested Class method");
        }
    }

    // Inner Class
    public class InnerClass {
        public void display() {
            System.out.println("Inner Class method");
            System.out.println("Outer Data: " + outerData);
        }
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();

        // Creating an instance of NestedClass
        OuterClass.NestedClass nested = new OuterClass.NestedClass();
        nested.display();

        // Creating an instance of InnerClass
        OuterClass.InnerClass inner = outer.new InnerClass();
        inner.display();
    }
}
```

Benefits of Nested classes

1. These classes provide a logical grouping of classes that have strong relationship with each other. We will cover the concepts of coupling and cohesion in week 6 to discuss this aspect further.
2. It increases encapsulation. This concept will be discussed in week 6 again.

3. Nesting small classes within top-level classes places the code closer to where it is used, thus improving the readability and maintainability of the code - a core aspect of software design.

Relevant Resources

Additional Reading Resources

- WALTER, S. Absolute Java, Global Edition. [Harlow]: Pearson, 2016. (Chapter 5)
- SCHILDT, H. Java: The Complete Reference, 12th Edition: McGraw-Hill, 2022 (Chapter 6 and 7)
- Classes and Objects (accessible on 14-02-2024) Oracle's Java Documentation. Available at: https://docs.oracle.com/javase/tutorial/java/javaOO/index.html