Lesson 53: Static Fields and Methods (W17D3)

Balboa High School

Michael Ferraro

December 9, 2015

Do Now

1. Create a new project called `Lesson53`.
2. Import `Textbook.java` from here.
3. Add a *symbolic constant*[1] to the `Textbook` class:
   `public int MAX_BOOKS_PER_PUPIL = 8`
4. Create a driver class called `TextbookDriver` with a `main()` creating two `Textbooks` — `t1` & `t2` — with titles "Java Methods" and "Physics," respectively.
5. After instantiating `Textbook` into the two instances specified above, add the following to `main()`:

```
System.out.println(t1.MAX_BOOKS_PER_PUPIL);
System.out.println(t2.MAX_BOOKS_PER_PUPIL);
```

[1] Remember to use the Java keyword that guarantees the value of this variable cannot change!

Aim

Students will learn the use of `static` for fields and methods in classes. Also, students will be introduced to the `Time` problem in PS #9.

What Remains in PS #9?

- §11, `Time` Class — overview at end of this lesson
- §12, Overloaded Methods — already talked about these
- §13, What is `static`? — explained in first part of this lesson
- §14, `SnackBar`, Part II — related to the use of `static`, overview of problem provided later in this lesson
- §15, Wrap-Up Exercises
- §16, Bonus Exercise — *for those who finish early. . .*

Are We Wasteful?

- Since `MAX_BOOKS_PER_PUPIL` is a symbolic constant, its value can never change. So. . . does each instance of `Textbook` really need its own copy of that `int`?

[2] Recall that an `int` in Java occupies 32 bits, or 4 bytes.

Are We Wasteful?

- Since `MAX_BOOKS_PER_PUPIL` is a symbolic constant, its value can never change. So. . . does each instance of `Textbook` really need its own copy of that `int`?
- If we had 500,000 instances of `Textbook` in memory, and each had its own `int MAX_BOOKS_PER_PUPIL`[2], how much memory is needed to accommodate this constant?

[2] Recall that an `int` in Java occupies 32 bits, or 4 bytes.

Are We Wasteful?

- Since `MAX_BOOKS_PER_PUPIL` is a symbolic constant, its value can never change. So. . . does each instance of `Textbook` really need its own copy of that `int`?
- If we had 500,000 instances of `Textbook` in memory, and each had its own `int MAX_BOOKS_PER_PUPIL`[2], how much memory is needed to accommodate this constant?

$$500{,}000 \text{ copies} \times 4 \frac{\text{bytes}}{\text{copy}}$$

2,000,000 bytes, or 2MB

[2] Recall that an `int` in Java occupies 32 bits, or 4 bytes.

Are We Wasteful?

- Consider a **large** production application, like Facebook, for which there might be $\approx 10^9$ accounts/profiles. Memory savings could mean...

Are We Wasteful?

- Consider a **large** production application, like Facebook, for which there might be $\approx 10^9$ accounts/profiles. Memory savings could mean...
  - fewer servers (lower overall memory requirement)
  - less rack space at a colocation center
  - less energy consumption!
  - see here

Are We Wasteful?

- Consider a **large** production application, like Facebook, for which there might be $\approx 10^9$ accounts/profiles. Memory savings could mean...
  - fewer servers (lower overall memory requirement)
  - less rack space at a colocation center
  - less energy consumption!
  - see here
- How can we have only one copy for such a variable (or constant in the case of `MAX_BOOKS_PER_PUPIL`)?

Are We Wasteful?

- Consider a **large** production application, like Facebook, for which there might be $\approx 10^9$ accounts/profiles. Memory savings could mean...
  - fewer servers (lower overall memory requirement)
  - less rack space at a colocation center
  - less energy consumption!
  - see here
- How can we have only one copy for such a variable (or constant in the case of `MAX_BOOKS_PER_PUPIL`)?

```
public static final int MAX_BOOKS_PER_PUPIL = 8;
```

Another Reason for One Copy

- Sometimes, we want there to be a single instance of a variable across multiple instances of a class.

Another Reason for One Copy

- Sometimes, we want there to be a single instance of a variable across multiple instances of a class.
- Consider this: Having a variable to keep track of the number of `Textbooks` out there would be useful. How can we keep count?

Another Reason for One Copy

- Sometimes, we want there to be a single instance of a variable across multiple instances of a class.
- Consider this: Having a variable to keep track of the number of `Textbooks` out there would be useful. How can we keep count?
- Solution: Declare a `static int` in `Textbook` to keep track. Each time a new `Textbook` is constructed, increase the value of this variable by 1.

Another Reason for One Copy

- Sometimes, we want there to be a single instance of a variable across multiple instances of a class.
- Consider this: Having a variable to keep track of the number of `Textbooks` out there would be useful. How can we keep count?
- Solution: Declare a `static int` in `Textbook` to keep track. Each time a new `Textbook` is constructed, increase the value of this variable by 1.
- Implement that solution now! Once done, modify the printed output to show the number of `Textbooks` in existence at each of these times:

Another Reason for One Copy

- Sometimes, we want there to be a single instance of a variable across multiple instances of a class.
- Consider this: Having a variable to keep track of the number of `Textbooks` out there would be useful. How can we keep count?
- Solution: Declare a `static int` in `Textbook` to keep track. Each time a new `Textbook` is constructed, increase the value of this variable by 1.
- Implement that solution now! Once done, modify the printed output to show the number of `Textbooks` in existence at each of these times:
  - after the "Java Methods" `Textbook` is created,

Another Reason for One Copy

- Sometimes, we want there to be a single instance of a variable across multiple instances of a class.
- Consider this: Having a variable to keep track of the number of `Textbooks` out there would be useful. How can we keep count?
- Solution: Declare a `static int` in `Textbook` to keep track. Each time a new `Textbook` is constructed, increase the value of this variable by 1.
- Implement that solution now! Once done, modify the printed output to show the number of `Textbooks` in existence at each of these times:
  - after the "Java Methods" `Textbook` is created,
  - after "Physics" has been created, and

Another Reason for One Copy

- Sometimes, we want there to be a single instance of a variable across multiple instances of a class.
- Consider this: Having a variable to keep track of the number of `Textbooks` out there would be useful. How can we keep count?
- Solution: Declare a `static int` in `Textbook` to keep track. Each time a new `Textbook` is constructed, increase the value of this variable by 1.
- Implement that solution now! Once done, modify the printed output to show the number of `Textbooks` in existence at each of these times:
  - after the "Java Methods" `Textbook` is created,
  - after "Physics" has been created, and
  - before any instances of `Textbook` yet exist (how?).

Tracking # of Textbooks

```
public class Textbook {

    // instance var (AKA field) ////////////////////////
    public String title;

    // class vars (i.e., one for entire class!) ////////
    public static final int MAX_BOOKS_PER_PUPIL = 8;
    public static int numBooks;

    public Textbook(String title) {
        this.title = title;
        numBooks++;
    }

}
```

Tracking # of Textbooks

```
public class TextbookDriver {

    public static void main(String[] args) {

        System.out.println("before t1, numBooks = "
            + Textbook.numBooks); //we'll discuss
                                  //usage shortly

        Textbook t1 = new Textbook("Java Methods");
        System.out.println("after t1, numBooks = "
            + t1.numBooks);

        Textbook t2 = new Textbook("Physics");
        System.out.println("after t2, numBooks = "
            + t2.numBooks);
    }
}
```

Ways To Access `static` Vars

When using *dot notation* to access a `static` variable, there are two options:[3]

- Instance-name-dot-variable-name:

  `t1.numBooks`

- Class-name-dot-variable-name:

  `Textbook.numBooks` ← **preferred way**

[3] . . . assuming the class accessing the variable has access per `public/private`!

Ways To Access static Vars

You've used the *class-name-dot-variable-name* method for accessing a class variable before!

```
public class MathTest {

    public static void main(String[] args) {

        //accessing class var using class name
        System.out.println( Math.PI );

    }
}
```

`static` Isn't Just for Vars!

- Recall what a class is: It's a definition that includes

`static` Isn't Just for Vars!

- Recall what a class is: It's a definition that includes
  - variables (sometimes constant)
  - methods (sometimes special ones, like constructors)

`static` Isn't Just for Vars!

- Recall what a class is: It's a definition that includes
  - variables (sometimes constant)
  - methods (sometimes special ones, like constructors)
- We just used `static` for `numBooks`, turning what otherwise would have been an *instance variable* into a *class variable*.

`static` Isn't Just for Vars!

- Recall what a class is: It's a definition that includes
  - variables (sometimes constant)
  - methods (sometimes special ones, like constructors)
- We just used `static` for `numBooks`, turning what otherwise would have been an *instance variable* into a *class variable*.
- We've seen `static` for methods plenty of times. Just consider `main()`:

```
public static void main(String[] args)
```

`static` Isn't Just for Vars!

- Recall what a class is: It's a definition that includes
  - variables (sometimes constant)
  - methods (sometimes special ones, like constructors)
- We just used `static` for `numBooks`, turning what otherwise would have been an *instance variable* into a *class variable*.
- We've seen `static` for methods plenty of times. Just consider `main()`:

```
public static void main(String[] args)
```

- No matter how many instances of a class are created, there will only ever be 1 `main()`! Why would more copies of those instructions in memory be useful, anyhow?

`static` Methods

- When is it appropriate to mark a method `static`?

`static` Methods

- When is it appropriate to mark a method `static`?
- Answer: When a method is a standalone "machine," taking arguments and returning something.

`static` Methods

- When is it appropriate to mark a method `static`?
- Answer: When a method is a standalone "machine," taking arguments and returning something.
- Example #1:

```
public class MathTest {

    public static void main(String[] args) {

        //call static method via class name
        System.out.println( Math.sqrt(2) );

    }
}
```

`static` Methods

- Consider this advantage of a `static` method: You need not instantiate a class into an object in order to call the method.

`static` Methods

- Consider this advantage of a `static` method: You need not instantiate a class into an object in order to call the method.

- So when we call `Math.sqrt()`, here's what we **didn't** have to do:

```
public class MathTest {

    public static void main(String[] args) {

        //call to static method via object
        Math mathObj = new Math();
        System.out.println( mathObj.sqrt(2) );

    }
}
```

static Methods

```
public class MathTest {

    public static void main(String[] args) {

        //call to static method via object
        Math mathObj = new Math();
        System.out.println( mathObj.sqrt(2) );

    }
}
```

- In fact, the above **doesn't even work** in the case of class `Math` — Oracle decided to make the constructor for that class `private`, so we don't even have the ability to create `mathObj`; we *must* call all `Math` class methods using the class-name-dot-method convention.

static Methods

- Example #2:

```
public class CircleTools {

    public static final double PI = 3.1415926;

    public static double getCircum(double r) {
       //returns circumference for circle
       //whose radius is r
       return 2 * PI * r;
    }
}
```

- What happens when `double PI` is no longer `static`? *Try!*

Overview: `SnackBar`, Part II

- When you reach Part II of `SnackBar` in PS #9, you'll be asked to implement a very important missing piece. . . the money counter!

Overview: `SnackBar`, Part II

- When you reach Part II of `SnackBar` in PS #9, you'll be asked to implement a very important missing piece. . . the money counter!
- What ideas do you now have for keeping track of how much money has been made across the three `Vendor` objects?

Overview: `SnackBar`, Part II

- When you reach Part II of `SnackBar` in PS #9, you'll be asked to implement a very important missing piece. . . the money counter!
- What ideas do you now have for keeping track of how much money has been made across the three `Vendor` objects?
- The Litvin §10.12 reading for `SnackBar`, Part II, is here.

Overview: `Time` Class

You'll write a class to...

- represent time in European/military/24-hour form

[4] Browse to here for an example of `throwing` and `catching` an exception.

Overview: `Time` Class

You'll write a class to...

- represent time in European/military/24-hour form
  - 3:07AM = 03:07 $\rightarrow$ `new Time(3,7);`

[4] Browse to here for an example of `throwing` and `catching` an exception.

Overview: `Time` Class

You'll write a class to...

- represent time in European/military/24-hour form
  - 3:07AM = 03:07 $\rightarrow$ `new Time(3,7);`
  - 8:24PM = 20:24 $\rightarrow$ `new Time(20,24);`

[4] Browse to here for an example of `throwing` and `catching` an exception.

Overview: `Time` Class

You'll write a class to...

- represent time in European/military/24-hour form
  - 3:07AM = 03:07 → `new Time(3,7);`
  - 8:24PM = 20:24 → `new Time(20,24);`
  - 12:19AM = 00:19 → `new Time(0,19);`

[4] Browse to here for an example of `throwing` and `catching` an exception.

Overview: `Time` Class

You'll write a class to...

- represent time in European/military/24-hour form
  - 3:07AM = 03:07 → `new Time(3,7);`
  - 8:24PM = 20:24 → `new Time(20,24);`
  - 12:19AM = 00:19 → `new Time(0,19);`
  - 12:30PM = 12:30 → `new Time(12,30);`

[4] Browse to here for an example of `throwing` and `catching` an exception.

Overview: `Time` Class

You'll write a class to...

- represent time in European/military/24-hour form
  - 3:07AM = 03:07 → `new Time(3,7);`
  - 8:24PM = 20:24 → `new Time(20,24);`
  - 12:19AM = 00:19 → `new Time(0,19);`
  - 12:30PM = 12:30 → `new Time(12,30);`
- throw an exception for construction of invalid times; e.g., `new Time(22,60)` and `new Time(24,-5)`

  `throw new IllegalArgumentException("InvalidTime");`[4]

[4] Browse to here for an example of `throwing` and `catching` an exception.

Overview: `Time` Class

You'll write a class to...

- represent time in European/military/24-hour form
  - 3:07AM = 03:07 → `new Time(3,7);`
  - 8:24PM = 20:24 → `new Time(20,24);`
  - 12:19AM = 00:19 → `new Time(0,19);`
  - 12:30PM = 12:30 → `new Time(12,30);`
- throw an exception for construction of invalid times; e.g., `new Time(22,60)` and `new Time(24,-5)`

  `throw new IllegalArgumentException("InvalidTime");`[4]
- convert time to # of minutes since 00:00 of same day via method you write: `toMins()`

[4] Browse to here for an example of `throwing` and `catching` an exception.

Overview: `Time` Class

You'll write a class to...

- determine whether the current `Time` object represent an earlier time than another `Time` object that is sent to `lessThan()`; e.g.,

```
Time t1 = new Time(5,09); //5:09AM
Time t2 = new Time(23,59); //11:59PM

boolean isT1Earlier = t1.lessThan(t2); //isT1Earlier
should be true
```

  Consider how `toMins()` could be useful when writing `lessThan()`.

Overview: `Time` Class

You'll write a class to...

- determine time difference, in minutes, between two times; e.g.,

```
Time t1 = new Time(13,30);   //1:30PM
Time t2 = new Time(9,30);    //9:30AM

System.out.println(t1.elapsedSince(t2));
//should print 240
```

Overview: `Time` Class

You'll write a class to...

- determine time difference, in minutes, between two times; e.g.,

```
Time t1 = new Time(13,30);   //1:30PM
Time t2 = new Time(9,30);    //9:30AM

System.out.println(t1.elapsedSince(t2));
//should print 240
```

- If you send `elapsedSince()` a *later* time than "this" time, you have to assume that the later time happened the day before!; e.g., what should be the result of running `t2.elapsedSince(t1)`? (It might be useful to visualize the `Times` on a timeline!)

Overview: `Time` Class

Find the question's text posted here.

Next. . .

Work on the remaining parts of PS #9.

- It is recommended that you finish the parts in order (though it's not absolutely necessary)
- PS #9 is due on **Monday, 14 December 2015** at the start of 5th Period. **Get as much done as soon as possible!**

HW

Finish PS #9! Make sure you budget your time between now and the due date, taking into account studying for finals.