Unit 5: Writing Classes

Mr. Pelletier, Lord Byng Secondary

Unit 5 Overview

In Unit 5, we explore in-depth creating classes for use in object oriented programming.

Topics will include:

- The `public` and `private` keywords and where to use them.
- The concepts of encapsulation and information hiding.
- Defining and initializing an object's state with constructors, including default constructors, overloaded constructors, and copy constructors
- We will learn how to use Javadoc comments.
- We will create accessor, mutator, and helper methods.
- We will revisit `static` methods and learn about `static` variables.
- We will revisit the concept of variable scope with regard to OOP
- We will learn about the `this` keyword.

5.1: Anatomy of a Class

Mr. Pelletier, Lord Byng Secondary

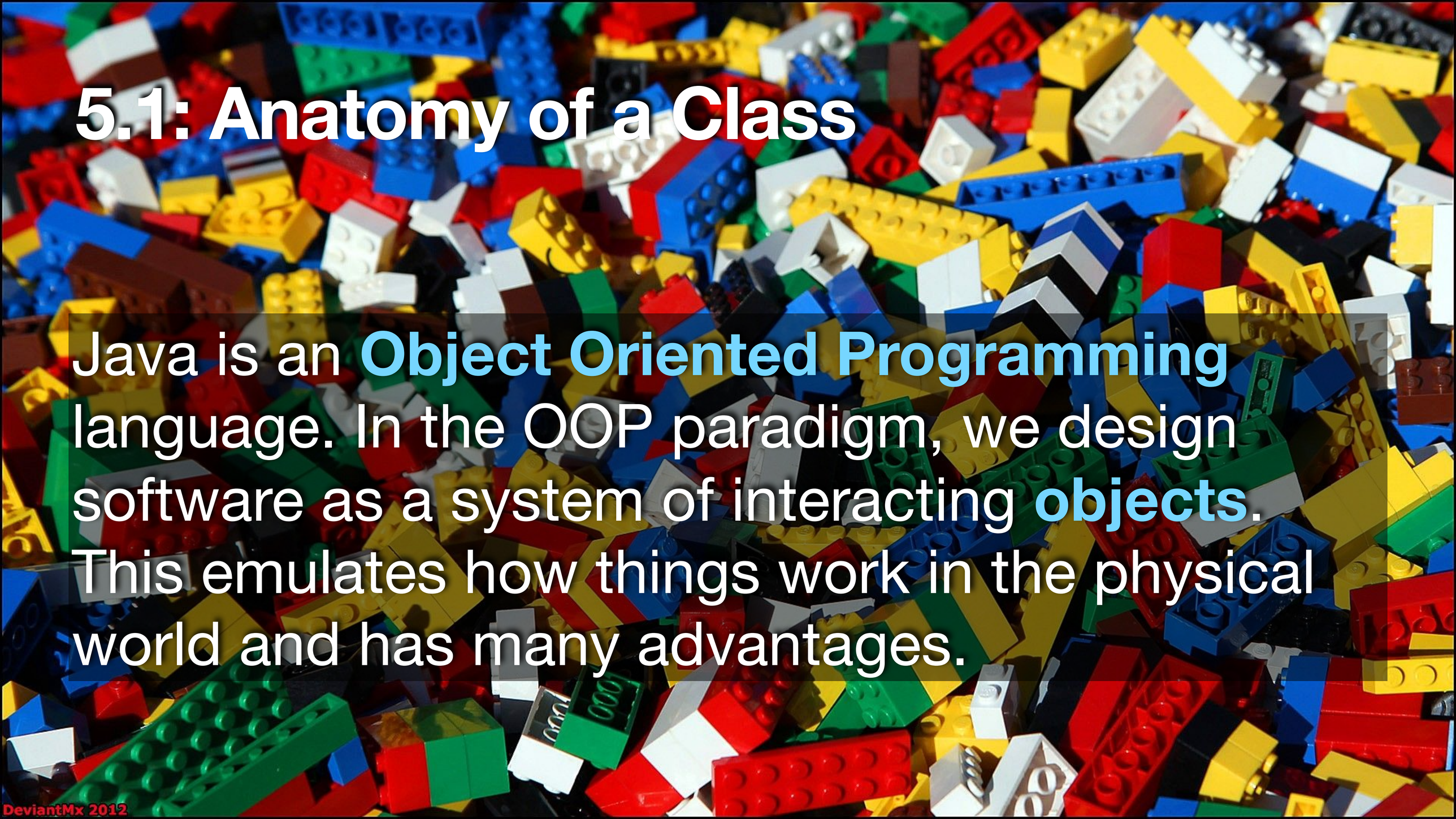
5.1: Anatomy of a Class

In 5.1, we will review classes and remind ourselves of their components.

We will formally discuss the `private` and `public` keywords, and introduce the concepts of encapsulation and information hiding.

5.1: Anatomy of a Class

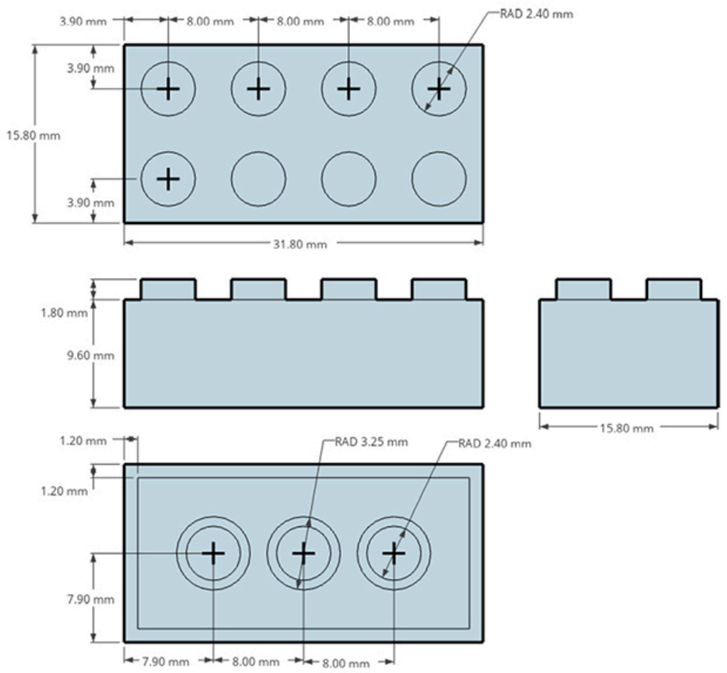
Java is an **Object Oriented Programming** language. In the OOP paradigm, we design software as a system of interacting **objects**. This emulates how things work in the physical world and has many advantages.



5.1 Anatomy of a Class

Classes vs. Objects

A **class** is a blueprint for an individual object. It describes the characteristics of objects created from it.



Objects are virtual entities that are created as **instances** of a class. Objects are the building blocks of an OOP program.

5.1 Anatomy of a Class

Class Definition Example

```
class Dog {

  int age;
  String name;

  Dog(int a, String n) {
    age = a;
    name = n;
  }

  void bark() {
    System.out.println(name + " says woof!");
  }
}
```

Attributes are stored in Instance Variables

Constructors initialize attributes

Behaviour Methods allow objects to act and interact

5.1 Anatomy of a Class

Encapsulation

Those are the basics. Now we want to dive deeper into Object Oriented Programming.

Our goal is to make classes **ENCAPSULATE** the complexity of the data and tasks they are associated with *inside* of objects.

As we enter this discussion, imagine yourself as a programmer who is making a class that will go into somebody else's code. That somebody else we will refer to as "the user."

5.1 Anatomy of a Class

Encapsulation

“Objects are like people. They’re living, breathing things that have knowledge inside them about how to do things and have memory inside them so they can remember things. And rather than interacting with them at a very low level, you interact with them at a very high level of abstraction, like we’re doing right here.”

5.1 Anatomy of a Class

Encapsulation

“Here’s an example: If I’m your laundry object, you can give me your dirty clothes and send me a message that says, “Can you get my clothes laundered, please.” I happen to know where the best laundry place in San Francisco is. And I speak English, and I have dollars in my pockets. So I go out and hail a taxicab and tell the driver to take me to this place in San Francisco. I go get your clothes laundered, I jump back in the cab, I get back here. I give you your clean clothes and say, “Here are your clean clothes.”

5.1 Anatomy of a Class

Encapsulation

“You have no idea how I did that. You have no knowledge of the laundry place. Maybe you speak French, and you can’t even hail a taxi. You can’t pay for one, you don’t have dollars in your pocket. Yet I knew how to do all of that. And you didn’t have to know any of it. All that complexity was hidden inside of me, and we were able to interact at a very high level of abstraction. That’s what objects are. They encapsulate complexity, and the interfaces to that complexity are high level.”

5.1 Anatomy of a Class

Levels of Abstraction 1/3

High Level

“Go Brush Your Teeth”

5.1 Anatomy of a Class

Levels of Abstraction 2/3

Low Level

Step 1
Brush at a 45 degree angle to your teeth. Direct the bristles to where your gums and teeth meet. Use a gentle, circular, massaging motion, up and down. Don't scrub. Gums that recede visibly are often a result of years of brushing too hard.

Step 2
Clean every surface of every tooth. The chewing surface, the cheek side, and the tongue side.

Step 3
Don't rush your brush. A thorough brushing should take at least two to three minutes. Try timing yourself

Step 4
Change your usual brushing pattern. Most people brush their teeth the same way all the time. That means they miss the same spots all the time. Try reversing your usual pattern.

Step 5
Use a soft brush with rounded bristles. The right toothbrush cleans better. Choose a size and shape that allow you to reach all the way to your back teeth. There are many different types of brushes, so ask your dentist to suggest the best one for you. CDA recommends you replace your toothbrush every three months.

5.1 Anatomy of a Class

Levels of Abstraction 3/3

Question: What is an example of interacting with the String class at a high level of abstraction?

5.1 Anatomy of a Class

Connecting 2.1 and 5.1

Question: Let’s imagine we’re making a ToothBrush class that encapsulates the five-step processing of brushing teeth. How do Instance Variables, Constructors, and Behaviour methods tie into this?

5.1 Anatomy of a Class

Encapsulation

Advantages: By breaking a complex system (eg: iOS, Minecraft) into an environment of encapsulated, interacting objects, you

Intuitive to plan: what objects do we need and how do they interact?

Easy to Develop in Teams: Who will write which class? How do I interact with other classes?

Reduces Bugs & Errors: individual objects can be made fail-proof, bugs can be traced to their source quicker.

5.1 Anatomy of a Class

Public and Private 1/2

In order to make encapsulation work, we have to make sure nobody can circumvent our design. We can do this by hiding the complexity behind a barrier using the **public** and **private** keywords.

Instance Variables and Behaviour Methods and can be made **public** or **private** to control access outside of the class.

5.1 Anatomy of a Class

Public and Private 2/2

Constructors are always public, and Instance Variables are almost always private.

What we make public defines how a user of our will interact will interact with instances of our Class.

5.1 Anatomy of a Class

A Better Dog Class 1/4

```
class Dog {

  int age;
  String name;

  Dog(int a, String n) {
    age = Math.max(a,0);
    name = n;
  }

  void bark() {
    System.out.println(name + " says woof!");
  }
}
```

Let's assume one of your goals in writing the dog class is to make sure Dog objects can't have a negative age. Why isn't this enough?

5.1 Anatomy of a Class

A Better Dog Class 2/4

```
Dog fluffy = new Dog(1, “Fluffy”);
fluffy.age = -99;
```

Outside of the class we can still access the age instance variable from the instance itself!

5.1 Anatomy of a Class

A Better Dog Class 3/4

Declaring the instance variables as private helps to prevent users from manually assigning a negative value.

```
public class Dog {

  private int age;
  private String name;

  public Dog(int a, String n) {
    age = Math.max(a,0);
    name = n;
  }

  public void bark() {
    System.out.println(name + " says woof!");
  }
}
```

5.1 Anatomy of a Class

A Better Dog Class 4/4

```
Dog fluffy = new Dog(1, “Fluffy”);
fluffy.age = -99;
```

If age is declared as private, this will not compile!

5.2: Constructors

Mr. Pelletier, Lord Byng Secondary

5.2: Constructors

Overview

Constructors initialize an object’s instance variables when an object of this class is constructed.

They have special naming conventions.

There are multiple types (Default, Parameterized, Copy).

If you have multiple types in your class, your constructors are Overloaded

5.2: Constructors

Naming Conventions

Must have the same name as the class, has no return type (not even void), and is always public. Eg:

```
public class LegoBrick {

  private int length;
  private int height;

  public LegoBrick() {
      //implementation not shown…
  }

}
```

5.2: Constructors

Types of Constructors 1/3

Default Constructors take no parameters. The programmer decides the instance variable's values. Eg:

```
public LegoBrick() {
  length = 1;
  height = 1;
}
```

5.2: Constructors

Types of Constructors 2/3

Parameterized Constructors take parameters and uses them to initialize the instance variable's values. Eg:

```
public LegoBrick(int L, int H) {
  length = L;
  height = H;
}
```

5.2: Constructors

Types of Constructors 3/3

Copy Constructors take an object of this class as a parameter and copies it's instance variable values. Eg:

```
public LegoBrick(LegoBrick myBrick) {
  length = myBrick.length;
  height = myBrick.height;
}
```

5.2: Constructors

Overloading

Constructors are considered **Overloaded** if you have more than one of them in your class. In order for this to work, each constructor must have different sets of parameters.

5.2: Constructors

Using a Constructor

Constructors are called outside of the class with the new keyword to create objects. If constructors are overloaded, you can select the constructor by the parameters you send. For example:

```
LegoBrick b1 = new LegoBrick(); //default

LegoBrick b2 = new LegoBrick(6,1); //parameterized

LegoBrick b3 = new LegoBrick(b2); //copy
```

5.3: Documenting Code

Mr. Pelletier, Lord Byng Secondary

5.3: Documenting Code

Overview

Javadoc comments can be used to auto-generate HTML documentation files for a class. They use this format and can span multiple lines:

```
/** Javadoc comment */
```

There are specific tags and forms you can use to shape how this is generated.

Most of all, you need to be able to read Javadoc comments as important information in Free Response questions will be given to you in this manner.

5.2: Constructors

Pre- and Post-condition

These are comments for methods. Put these first before any tags:

Pre-conditions: Describes what assumptions your code is making before the method runs. These assumptions might include how you expect parameter data to be formatted or constraints of their possible values. For example: If you method assumes a String will not be null, declare that here.

Post-conditions: Describes conditions that your code imposes after the method runs. For example: Fraction constructors force denominators to be non-zero.

5.2: Constructors

Tags

These go at the start of your java file:

`@author` - A place for the author of the class to sign their name

`@version` - What version of the class is this?

These are tied to each method and go after pre- and post-conditions:

`@param` - Describes individual parameters of a method. Include multiple @param tags for each parameter

`@return` - Describe what the method returns.

5.4: Accessor Methods

Mr. Pelletier, Lord Byng Secondary

5.4: Accessor Methods

Overview 1/2

If we make instance variables private, there is no way for **client code** (code outside the class) to look at the values of those variables.

We create accessor methods that allow client code to look at but not change those values.

5.4: Accessor Methods

Overview 2/2

Accessor methods are sometimes called “getters” because they “get” the instance variables values.

These methods are always public and never have a void return type.

5.4: Accessor Methods

Accessors for Dog? 1/2

```
public class Dog {

  private int age;
  private String name;

  public Dog(int a, String n) {
    age = Math.max(a,0);
    name = n;
  }

  public void bark() {
    System.out.println(name + “ says woof!”);
  }
}
```

5.4: Accessor Methods

Accessors for Dog? 2/2

```
public int getAge() {
   return age;
}

public String getName() {
   return name;
}
```

5.4: Accessor Methods

Special Accessors

The `toString()` method is a special method that is automatically called whenever an object appears in a context where a String is expected. It converts an Object into a String format. For example, dog's toString would look like this:

```
public String toString() {
  return name + “, age: “ + age;
}
```

5.4: Accessor Methods

Special Accessors

If you made a Dog object like this:

```
Dog fluffy = new Dog(3, “Fluffy”);
```

And then printed it like this:

```
System.out.println(fluffy);
```

Then the toString method will automatically run and this text will be printed to the console:

```
Fluffy, age 3
```

5.5: Mutator Methods

Mr. Pelletier, Lord Byng Secondary

5.5: Mutator Methods

Overview 1/2

If we make instance variables private, there is no way for **client code** (code outside the class) to change at the values of those variables.

We create mutator methods that allow client code to change those values.

5.5: Mutator Methods

Overview 2/2

You might think, “Wait, why did we make these variables private if we’re just going to let people modify them through mutator methods?”

The reason is that Mutator methods can be programmed to enforce rules. You can modify the variables, but the Mutator method can prevent client code from modifying in bad ways (such as setting a Dog’s age to a negative number).

5.5: Mutator Methods

Mutators for Dog? 1/2

```
public class Dog {

  private int age;
  private String name;

  public Dog(int a, String n) {
    age = Math.max(a,0);
    name = n;
  }

  public void bark() {
    System.out.println(name + “ says woof!”);
  }
}
```

5.5: Mutator Methods

Mutators for Dog? 2/2

```
public void setAge(int a) {
   age = Math.max(a,0);
}

public void setName(String n) {
   if (n == null)
      name = "";
   else
      name = n;
}
```

5.6: Writing Methods

Mr. Pelletier, Lord Byng Secondary

5.7: Static Methods and Variables

Mr. Pelletier, Lord Byng Secondary

5.7: Static Methods and Variables

Overview

The static keyword means a method or variable belongs to the class, not objects of that class.

It does not mean "unchanging" or "frozen," even though that is what the word static often means in English.

5.7: Static Methods and Variables

Static or Class Variables

If an instance variable is declared as static, that means it is a "class variable" and belongs to the class, not objects of that class.

Its value is shared across all objects of that class.

5.7: Static Methods and Variables

Static or Class Methods

If a method is declared as static, that means it is a "class method" and it is called from the class, not an object.

Static methods can’t access or modify instance variables, but can access and modify class variables.

5.7: Static Methods and Variables

Examples

```
class StoreItem {

  private double price;
  private static int taxRate = 0.10;

  public StoreItem(int p) {
    price = p;
  }

  public double getTotalPrice() {
    return price + price*taxRate;
  }

}
```

5.7: Static Methods and Variables

Examples

Note how in this example, mountainDew and redBull objects each have a price, but the static tax variable is the same for all StoreItem objects:

```
StoreItem mountainDew = new StoreItem(2.00);
StoreItem redBull = new StoreItem(2.50);
System.out.println(mountainDew.getTotalPrice());
System.out.println(redBull.getTotalPrice());
```

5.7: Static Methods and Variables

Examples

Note how in this example, mountainDew and redBull objects each have a price, but the static tax variable is the same for all StoreItem objects:

```
StoreItem mountainDew = new StoreItem(1.00);
StoreItem redBull = new StoreItem(2.50);
System.out.println(mountainDew.getTotalPrice());
System.out.println(redBull.getTotalPrice());
```

What if we wanted to make a method to change tax?

5.7: Static Methods and Variables

Examples

```
class StoreItem {

  private double price;
  private static int taxRate = 0.10;

  public StoreItem(int p) {
    price = p;
  }

  public double getTotalPrice() {
    return price + price*taxRate;
  }

  public static void changeTax(int t) {
    taxRate = t;
  }

}
```

Methods that access or modify static variables must also be static.

5.7: Static Methods and Variables

Examples

Now what is the result here?

```
StoreItem mountainDew = new StoreItem(2.00);
StoreItem redBull = new StoreItem(2.50);
StoreItem.changeTax(0.20);
System.out.println(mountainDew.getTotalPrice());
System.out.println(redBull.getTotalPrice());
```

5.7: Static Methods and Variables

Examples

See how it is called from the Class?

```
StoreItem mountainDew = new StoreItem(2.00);
StoreItem redBull = new StoreItem(2.50);
StoreItem.changeTax(0.20);
System.out.println(mountainDew.getTotalPrice());
System.out.println(redBull.getTotalPrice());
```

It doesn't make sense to call `changeTax()` from an object because it is changing the tax for ALL objects.

5.7: Static Methods and Variables

Examples

```
class StoreItem {

  private double price;
  private static int taxRate = 0.10;

  public StoreItem(int p) {
    price = p;
  }

  public double getTotalPrice() {
    return price + price*taxRate;
  }

  public void changeTax(int t) {
    taxRate = t;
  }

}
```

What if we made it non-static?

5.7: Static Methods and Variables

Examples

See how it is called from the Class?

```
StoreItem mountainDew = new StoreItem(2.00);
StoreItem redBull = new StoreItem(2.50);
StoreItem.changeTax(0.20);
System.out.println(mountainDew.getTotalPrice());
System.out.println(redBull.getTotalPrice());
```

It doesn't make sense to call `changeTax()` from an object because it is changing the tax for ALL objects.

5.7: Static Methods and Variables

Examples

```
class StoreItem {

  private double price;
  private static int taxRate = 0.10;

  public StoreItem(int p) {
    price = p;
  }

  public static double getTotalPrice() {
    return price + price*taxRate;
  }

  public static changeTax(int t) {
    taxRate = t;
  }

}
```

Why can’t we made getTotalPrice static?

5.7: Static Methods and Variables

Examples

See how it is called from the Class?

```
StoreItem mountainDew = new StoreItem(2.00);
StoreItem redBull = new StoreItem(2.50);
StoreItem.getTotalPrice();
```

It doesn't make sense to call getTotalPrice() from the class … which object's total price are we trying to get???

This code won't even compile.

5.8: Scope and Access

Mr. Pelletier, Lord Byng Secondary

5.8: Scope and Access

Overview

In this lesson we will understand how **local variables** and **instance variables** interact.

5.8: Scope and Access

Instance and Class Variables

Instance Variables and Class variables are declared at the top of the class, and have scope throughout the entire class.

```
public class StoreItem {

    private double price;
    private Static double taxRate = 0.1;


    // etc…

}
```

5.8: Scope and Access

Local Variables

In addition, you can declare **local variables** within a single method. These include, but are not limited to, parameters.

Think back to your Fraction class. What are some examples of local variables? What is their purpose?

5.8: Scope and Access

Local Variables

Local variables are neither private, public, or static because they can’t be shared outside of the class and the belong to neither the class or the object.

The scope of a local variable is limited to the method in which they are declared.

When an instance and a local variable with the same name both have scope in a method, that name refers to the local variable

5.8: Scope and Access

Local Variables

```
class Blooper {
  private int x;

  public Blooper() {
    int y = 10;
    x = y;
  }

  public void setX(int newX) {
    private int x;
    x = newX;
    y++;
  }
```

Spot and fix the syntax errors.
Why isn't setX a mutator method?

5.8: Scope and Access

Examples 1/6

Consider the following class definition:

```
public class Example{
   private int current;

   public Example (int c){
      double factor = Math.random();
      current = (int)(c * factor);
   }

   public void reset(int num){
      double factor = Math.random();
      c += (int)(num * factor);
   }

   //other methods not shown
}
```

Which of the following is the reason this code does not compile?

(A) The **reset** method cannot declare a variable named **factor** since a variable named **factor** is declared in the constructor.

(B) The constructor cannot declare a variable named **factor** since a variable named **factor** is declared in the **reset** method.

(C) The constructor cannot access the instance variable **current** since **current** is **private**.

(D) The **reset** method cannot access the variable **c** since **c** is declared in the constructor as a parameter and not declared in the **reset** method.

(E) There is no syntax error in this code and it would compile.

5.8: Scope and Access

Examples 2/6

Consider the following class definition:

```
public class Example{
   private int current;

   public Example (int c){
      double factor = Math.random();
      current = (int)(c * factor);
   }

   public void reset(int num){
      double factor = Math.random();
      c += (int)(num * factor);
   }

   //other methods not shown
}
```

Which of the following is the reason this code does not compile?

(A) The `reset` method cannot declare a variable named `factor` since a variable named `factor` is declared in the constructor.

(B) The constructor cannot declare a variable named `factor` since a variable named `factor` is declared in the `reset` method.

(C) The constructor cannot access the instance variable `current` since `current` is `private`.

(D) The `reset` method cannot access the variable `c` since `c` is declared in the constructor as a parameter and not declared in the `reset` method.

(E) There is no syntax error in this code and it would compile.

5.8: Scope and Access

Examples 3/6

Consider the following class definition:

```
public class Example{
   private int current;

   public Example (int c){
      double factor = Math.random();
      current = (int)(c * factor);
   }

   public void reset(int num){
      private double factor = Math.random();
      current += (int)(num * factor);
   }

   //other methods not shown
}
```

Which of the following is the reason this code does not compile?

(A) The `reset` method cannot declare a variable named `factor` since a variable named factor is declared in the constructor.

(B) The `reset` method cannot declare `factor` as `private` since `factor` is a local variable not an instance variable.

(C) The constructor cannot declare a variable named `factor` since a variable named factor is declared in the `reset` method.

(D) The constructor cannot access the instance variable `current` since `current` is `private`.

(E) There is no syntax error in this code and it would compile.

5.8: Scope and Access

Examples 4/6

Consider the following class definition:

```
public class Example{
    private int current;

    public Example (int c){
        double factor = Math.random();
        current = (int)(c * factor);
    }

    public void reset(int num){
        private double factor = Math.random();
        current += (int)(num * factor);
    }

    //other methods not shown
}
```

Which of the following is the reason this code does not compile?

(A) The `reset` method cannot declare a variable named `factor` since a variable named factor is declared in the constructor.

(B) The `reset` method cannot declare `factor` as `private` since `factor` is a local variable not an instance variable.

(C) The constructor cannot declare a variable named `factor` since a variable named factor is declared in the `reset` method.

(D) The constructor cannot access the instance variable `current` since `current` is `private`.

(E) There is no syntax error in this code and it would compile.

5.8: Scope and Access

Examples 5/6

```
public class ExampleTwo{
    private int current;

    public ExampleTwo (int c){
        current = c;
    }

    public void reset(int amt){
        amt += 5;
        int current = amt;
    }

    public int getCurrent()
        return current;
    }
}
```

In a different class:

```
ExampleTwo obj = new ExampleTwo(10);
obj.reset(15);
System.out.println(obj.getCurrent());
```

What will be displayed as a result of executing the above code?

(A) 0

(B) 5

(C) 10

(D) 20

(E) The code will not compile because the **increase** method cannot declare a variable named **current**.

5.8: Scope and Access

Examples 6/6

```
public class ExampleTwo{
    private int current;

    public ExampleTwo (int c){
        current = c;
    }

    public void reset(int amt){
        amt += 5;
        int current = amt;
    }

    public int getCurrent()
        return current;
    }
}
```

In a different class:

```
ExampleTwo obj = new ExampleTwo(10);
obj.reset(15);
System.out.println(obj.getCurrent());
```

What will be displayed as a result of executing the above code?

(A) 0

(B) 5

(C) 10

(D) 20

(E) The code will not compile because the **increase** method cannot declare a variable named **current**.

5.9: `this` keyword

Mr. Pelletier, Lord Byng Secondary

5.9: this Keyword

Overview

In Java, when you have a local and instance variable overlap in scope, you can use the **`this`** keyword to improve readability.

It cannot be used in static methods.

5.9: this Keyword

Example: Reduce 1/2

Here is the reduce method (simplified) from Fraction using this:

```
public void reduce() {
    int gcf = GCF(this.numerator, this.denominator);
    this.numerator /= gcf;
    this.denominator /= gcf;
  }
}
```

5.9: this Keyword

Example: Reduce 2/2

In general, this refers to the object from which a non-static method was called. Using our Fraction class, if you called:

```
Fraction a = new Fraction("2/4");
Fraction b = new Fraction("1/3");
Fraction c = a.reduce();
Fraction d = b.reduce();
```

In this case, in the reduce method, this would refer to Fraction a

In this case, in the reduce method, this would refer to Fraction b

5.9: this Keyword

Improve Readability

```
public class Dog {
    private String breed;
    //other data members not shown

    public String getBreed(){
        return breed;
    }


    public boolean isSameBreed(Dog otherDog){

        return breed.equals(otherDog.breed);

    }

    //other methods not shown
}
```

5.9: this Keyword

Improve Readability

```
public class Dog {
    private String breed;
    //other data members not shown

    public String getBreed(){
        return breed;
    }


    public boolean isSameBreed(Dog otherDog){
        return breed.equals(otherDog.breed);
    }

    //other methods not shown
}
```

When `breed` is used with `otherDog`, it is clear which object's `breed` is being accessed.

5.9: this Keyword

Improve Readability

```
public class Dog {
    private String breed;
    //other data members not shown

    public String getBreed(){
        return breed;
    }


    public boolean isSameBreed(Dog otherDog){

        return this.breed.equals(otherDog.breed);

    }

    //other methods not shown
}
```

Using this with dot notation to access **breed** of the calling object can be used to make it clearer which object's **breed** is being accessed at the beginning of the statement.

5.9: this Keyword

Improve Readability

```
public class Dog {
    private String breed;
    private int age;
    private double weight;
    private double score;

    public Dog(String breed, int age, double weight, double score){
        breed = breed;
        age = age;
        weight = weight;
        score = score;
    }
```

5.9: this Keyword

Improve Readability

```
public class Dog {
    private String breed;
    private int age;
    private double weight;
    private double score;

    public Dog(String b, int a, double w, double s){
        breed = b;
        age = a;
        weight = w;
        score = s;
    }
```

5.9: this Keyword

Improve Readability

```
public class Dog {
    private String breed;
    private int age;
    private double weight;
    private double score;

    public Dog(String breed, int age, double weight, double score){
        this.breed = breed;
        this.age = age;
        this.weight = weight;
        this.score = score;
    }
```

5.9: this Keyword

this as a Parameter

```
public class DogCompetition {
    //data members not shown

    public static int awardsPoints(Dog competitor){
        if (isWinner(competitor))
            return 10;
        return -5;
    }

    public static boolean isWinner(Dog d){
        //implementation not shown
    }

    //other methods not shown
}
```

5.9: this Keyword

this as a Parameter

```
public class Dog {
    private String breed;
    private int age;
    private double weight;
    private double score;

    public void updateScore(){
        score += DogCompetition.awardPoints(this);
    }

    //other methods not shown
}
```

```
public static int awardsPoints(Dog competitor){
    if (isWinner(competitor))
        return 10;
    return -5;
}
```

5.9: this Keyword

Example

```
public class SampleOne{
   private int x;
   public SampleOne(int x){
      this.x = x;
   }
   public void increase(){
      this.x += SampleTwo.amount(this);
   }
   public int getX(){
      return this.x;
   }
}

public class SampleTwo{
   public static int amount(SampleOne s){
      if (s.getX() > 10)
         return 2;
      return 5;
   }
}
```

In a different class:

```
SampleOne obj1 = new SampleOne(33);
obj1.increase()
System.out.println(obj1.getX());
```

What is the output after executing the above code?

(A) 33
(B) 35
(C) 40
(D) The code has syntax error because `this` is used incorrectly in the `SampleOne` constructor.
(E) The code has a syntax error because `this` is used incorrectly in the `SampleOne` `increase()` method.