WIT COMP1000

Classes

Review: Scanner

- Recall that `Scanner` variables are used to get input from the keyboard (with `System.in`) or from files
- We use the `hasNext()`, `hasNextLine()`, `hasNextInt()`, and `hasNextDouble()` methods to check if there are more inputs
- Then we use the `next()`, `nextLine()`, `nextInt()`, and `nextDouble()` methods to read those inputs

Scanner Example

```
import java.io.File;
import java.io.FileNotFoundEx
import java.util.Scanner;

public class ClassExamples {
     public static void mai(String[] args) {
          Scanner keyboardIn = new Scanner(System.in
          System.out.print("Enter the file name:
          String inputFileName = keyboardIn.next();
          try (Scanner inputFile = new Scanner(new File(inputFileName))) {
               while(inputFile.hasNextL
                    String line = inputFile.nextL
                    System.out.println(
               }
          } catch (FileNotFoundExc
               System.out.printl
               System.exit(0
     }
}
```

keyboardIn is a Scanner variable

keyboardIn.next() used to read one String from System.in

inputFile is another Scanner variable

inputFile.hasNextLine() used to check if there are more lines in the file

inputFile.nextLine() used to read one line from the file

`Scanner` Methods

- The `next()`, `hasNextLine()`, and `nextLine()` methods are part of the Scanner *class*
  - Same with the other `Scanner` methods we've seen
- When calling any of the class methods, you must use one *instance* of a variable of that class, called an *object*, as the identifier before the method call
  - Generic form: `RETURN_VALUE = OBJECT.METHOD(ARGUMENTS);`
  - Specific example: `line = inputFile.nextLine();`

Object-Oriented Programming (OOP)

- A programming paradigm based on the concept of "objects", which commonly represent real world entities
  - » For example: a person, a car, a pencil, a sensor
- Objects have *data fields*, or attributes, that represent the state of the object
- Objects have *methods*, or actions, that use or modify the data fields of the object
- Examples of OO languages: C++, C#, Java, JavaScript, PHP, Python, Ruby, …

OOP Terminology

- *Classes* are like templates
  - » Identify the data fields and methods that every instance of this type will have
  - » Many Java packages include class definitions already, such as the `java.io` package
  - » You can define your own class types as well
- An *Object* is a specific instance of a class, i.e., it is a variable of the class type
  - » For example, variables of type `Rectangle` might be named `rectA` and `rectB`

Example Class: Rectangle

```
public class Rectangle {
    public double length;
    public double width;
}
```

- This defines a class named `Rectangle`
- We will discuss the meaning of the **`public`** keyword for the class and for the variables later
- The two **`double`** lines say that the `Rectangle` class uses two **`double`** data fields named `length` and `width`
  - » In other words, *every* instance of the `Rectangle` class (a Rectangle *object*) will have its own `length` and `width`

Data Fields

- Defining a data field in a class is NOT a variable declaration
- You can not use those variables except in the context of an instance of the class (an object)
- In other words, you can think of the data fields as sub-pieces of an object that you can only access as part of the object
- So, when you declare a variable of a class type, it automatically has all of the data fields for that object
- Using the data fields is similar to using any other variable of the same type

Creating a New Class

- To create a new class that is part of an existing project:
  - » Right click on the project heading in Eclipse
  - » Select `New`, and `Class`
  - » Give the class a name, e.g., `Rectangle`
  - » Don't worry about all of the other options for now
- You can create a class with a `main()` method (we've been doing it all semester!), but not all classes will have a `main()` method
  - » Often have only one class with a `main()` method in each project

One Class per File

- In Java, each class you define must go into a separate Java file in the project
- It is common to have a single class that has nothing in it but a `main()` method
  - Used to start the program
- You may have several classes (and hence several Java files) in a project
- We will follow this model

Using the Rectangle Class

```
public class ClassExamples {
    public static void main(String[] args) {
        Rectangle rectA = new Rectangle();
        rectA.length = 4.2;
        rectA.width = 10.0;
        System.out.printf("Rectangle length: %.3f%n", rectA.length);
        System.out.printf("Rectangle width: %.3f%n", rectA.width);
    }
}
```

Create an object named `rectA` of type `Rectangle`

Give `rectA`'s `length` data field a value of `4.2` and `rectA`'s `width` data field a value of `10.0`

Multiple Class Objects

- You can declare more than one variable of a class type
- Each instance of a class variable has its own data fields that are completely separate from other objects of the same type
- For example, declaring two `Rectangle` objects actually declares four `double` variables (two `length` variables and two `width` variables)

Example: Multiple `Rectangle` Objects

Rectangle.java:

```
public class Rectangle {
     public double length;
     public double width;
}
```

ClassExamples.java:

```
public class ClassExamples {
     public static void main(String[] args) {
          Rectangle rectA = new Rectangle();
          Rectangle rectB = new Rectangle();

          rectA.length = 4.2;
          rectA.width = 10.0;
          rectB.length = 3.8;
          rectB.width = 2.5;

          System.out.printf("Rectangle A length: %.3f%n", rectA.length);
          System.out.printf("Rectangle A width: %.3f%n", rectA.width);
          System.out.printf("Rectangle B length: %.3f%n", rectB.length);
          System.out.printf("Rectangle B width: %.3f%n", rectB.width);
     }
}
```

Create two `Rectangle` variables named `rectA` and `rectB`

Set the `length` and `width` data fields for `rectA`

Set the `length` and `width` data fields for `rectB`

Objects in Memory

- When an object is created, memory is allocated for every data field, for example:

```
Rectangle rectA = new Rectangle();
Rectangle rectB = new Rectangle();
rectA.length = 4.2;
rectA.width = 10.0;
rectB.length = 3.8;
rectB.width = 2.5;
```

address	value	variable
0xffa000	4.2	rectA.length
0xffa008	10.0	rectA.width
0xffa010	3.8	rectB.length
0xffa018	2.5	rectB.width
0xffa020		
0xffa028		
0xffa030		

...

Exercise

- Define a class named `Triangle`. It should have two data fields: base and height. Write a `main()` method in another class (your default class for lecture examples is fine) that declares a `Triangle` object, assigns values to the two data fields, and then prints out both data fields.

Answer

Triangle.java:

```
public class Triangle {
	public double base;
	public double height;
}
```

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		Triangle myTri = new Triangle();

		myTri.base = 17.348;
		myTri.height = 104.6;

		System.out.printf("Triangle base: %.3f%n", myTri.base);
		System.out.printf("Triangle height: %.3f%n", myTri.height);
	}
}
```

Class Methods

- In addition to data fields, classes can include class methods
- Class methods are like any other method, except that you have access to the data fields for the class while inside the method
  - » This allows you to take advantage of all of the data fields without having to pass them in as arguments
- They are called using one instance of the class, as we've seen with `Scanner` and `PrintWriter`

Defining Class Methods

```
public class Rectangle {
    public double length;
    public double width;
    public void print() {
        System.out.printf("length: %.3f, ", length);
        System.out.printf("width: %.3f%n", width);
    }
}
```

- We've added a class method, `print()`, that returns nothing, takes no arguments, and prints out the `length` and `width` values of the rectangle
  - » Note the lack of the **`static`** keyword – stay tuned
- Note that `length` and `width` are used because they are part of the same class as the method
  - » The `print()` method will be called as part of one object and it will therefore use `length` and `width` from that object

Example: Using Class Methods

Rectangle.java:

```
public class Rectangle {
	public double length;
	public double width;

	public void print() {
		System.out.printf("length: %.3f, ", length);
		System.out.printf("width: %.3f%n", width);
	}
}
```

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		Rectangle rectA = new Rectangle();
		Rectangle rectB = new Rectangle();

		rectA.length = 4.2;
		rectA.width = 10.0;
		rectB.length = 3.8;
		rectB.width = 2.5;

		System.out.print("Rectangle A: ");
		rectA.print();
		System.out.print("Rectangle B: ");
		rectB.print();
	}
}
```

Class Methods vs Non-Class Methods

- We could create a `print()` method that is not part of the class
  - » Like other methods we've been writing so far this semester
  - » Part of the same class as `main()`
- To print a `Rectangle`, we would have to pass in two parameters to the method
  - » Or, in general, as many parameters as there are data fields
- More importantly, by adding the `print()` method as a class method we are *encapsulating* all of the actions and information about the Rectangle within the class

Example: Don't Do This!

Rectangle.java:

```
public class Rectangle {
      public double length;
      public double width;
}
```

BadClassExamples.java:

```
public class BadClassExamples {
      public static void rectanglePrint(double length, double width) {
            System.out.printf("length: %.3f, ", length);
            System.out.printf("width: %.3f%n", width);
      }

      public static void main(String[] args) {
            Rectangle rectA = new Rectangle();
            Rectangle rectB = new Rectangle();

            rectA.length = 4.2;
            rectA.width = 10.0;
            rectB.length = 3.8;
            rectB.width = 2.5;

            System.out.print("Rectangle A: ");
            rectanglePrint(rectA.length, rectA.width);
            System.out.print("Rectangle B: ");
            rectanglePrint(rectB.length, rectB.width);
      }
}
```

Use Class Methods!

- If a method is utilizing or manipulating data fields that are part of a class, then that method should be a class method of that class
- If an action is logically associated with a particular kind of object, then make a method that is part of the class for those kinds of objects
- Aside: all methods are part of *some* class in Java
  - » Put methods where they belong "best"

Adding Another Class Method

Rectangle.java:

```
public class Rectangle {
	public double length;
	public double width;

	public double area() {
		return length * width;
	}

	public void print() {
		System.out.printf("length: %.3f, ", length);
		System.out.printf("width: %.3f%n", width);
	}
}
```

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		Rectangle rectA = new Rectangle();
		Rectangle rectB = new Rectangle();

		rectA.length = 4.2;
		rectA.width = 10.0;
		rectB.length = 3.8;
		rectB.width = 2.5;

		System.out.print("Rectangle A: ");
		System.out.printf("area: %.3f, ", rectA.area());
		rectA.print();
		System.out.print("Rectangle B: ");
		System.out.printf("area: %.3f, ", rectB.area());
		rectB.print();
	}
}
```

Exercise

- Add two class methods to your `Triangle` class: `print()` and `area()`. `print()` should print the values of the data fields. `area()` should return the area of the triangle. Update your `main()` method to test the additional methods.

Answer

Triangle.java:

```
public class Triangle {
	public double base;
	public double height;

	public double area() {
		return 0.5 * base * height;
	}

	public void print() {
		System.out.printf("base: %.3f, ", base);
		System.out.printf("height: %.3f%n", height);
	}
}
```

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		Triangle myTri = new Triangle();

		myTri.base = 17.348;
		myTri.height = 104.6;

		System.out.print("Triangle myTri: ");
		System.out.printf("area: %.3f, ", myTri.area());
		myTri.print();
	}
}
```

Calling Class Methods from the Class

- You can call class methods from other class methods of the same class
- When doing so, you use the method directly, just like when accessing data fields
- That is, there is no object name and a dot before the name of the method, because you are already in the context of the class
  - » This changes if you are referring to another instance of the class

Example: Calling One Class Method from Another

Rectangle.java:

```
public class Rectangle {
	public double length;
	public double width;

	public double area() {
		return length * width;
	}

	public void print() {
		System.out.printf("length: %.3f, ", length);
		System.out.printf("width: %.3f, ", width);
		System.out.printf("area: %.3f%n", area());
	}

}
```

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		Rectangle rectA = new Rectangle();
		Rectangle rectB = new Rectangle();

		rectA.length = 4.2;
		rectA.width = 10.0;
		rectB.length = 3.8;
		rectB.width = 2.5;

		System.out.print("Rectangle A: ");
		rectA.print();
		System.out.print("Rectangle B: ");
		rectB.print();
	}
}
```

Visibility Modifier

- Every data field and class method has a set *visibility* that determines which other classes (if any) are allowed to access that data field or invoke that class method
- There are four visibility levels: **`public`**, **`private`**, **`protected`**, and package-private
- We will only be using **`public`** and **`private`** in this course
  - » We'll touch on package-private shortly
  - » You will learn about **`protected`** when you learn about inheritance (if you take Computer Science II)

`public` Visibility

- So far, we've only been using `public` data fields and class methods
- `public` data fields and class methods can be accessed directly from anywhere in your program
- You can use `public` data fields just like any other variables (using the `OBJECT.FIELD` syntax)
- You can use `public` class methods just like any other methods (using the `OBJECT.METHOD()` syntax)

`private` Visibility

- **`private`** data fields and class methods can only be accessed and used inside of other methods of the same class
- **`private`** data fields can only be used inside class methods of the same class
- **`private`** class methods can only be called from other methods of the same class
- You can *NOT* use private data fields or class methods from methods outside of that class, such as `main()`

Example: Visibility

PlayingCard.java:

```
public class PlayingCard {
	private String mySuit;
	private int myRank;

	public void setCard(String suit, int rank) {
		mySuit = suit;
		myRank = rank;
	}

	public void print() {
		if (myRank == 13) {
			System.out.print("King");
		} else if (myRank == 12) {
			System.out.print("Queen");
		} else if (myRank == 11) {
			System.out.print("Jack");
		} else if (myRank == 1) {
			System.out.print("Ace");
		} else {
			System.out.print(myRank);
		}
		System.out.println(" of " + mySuit);
	}
}
```

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		PlayingCard c = new PlayingCard();
		c.setCard("Spades", 12);
		c.print();

		c.mySuit = "Hearts"; // compiler error
	}
}
```

mySuit is a private data field of PlayingCard, so it can only be used in methods in the PlayingCard class

Exercise

- Modify your `Triangle` class so that both the base and height data field are **`private`**. Then add two new **`public`** methods: one named `setBase()` to set the value for the base and one named `setHeight()` to set the value for the height. Update your `print()` method to also print the area of the triangle (using your `area()` method!). Finally, update your `main()` method to test these modifications.

Answer

Triangle.java:

```
public class Triangle {
	private double base;
	private double height;

	public void setBase(double newBase) {
		base = newBase;
	}

	public void setHeight(double newHeight) {
		height = newHeight;
	}

	public double area() {
		return 0.5 * base * height;
	}

	public void print() {
		System.out.printf("base: %.3f, ", base);
		System.out.printf("height: %.3f, ", height);
		System.out.printf("area: %.3f%n", area());
	}
}
```

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		Triangle myTri = new Triangle();

		myTri.setBase(17.348);
		myTri.setHeight(104.6);

		System.out.print("Triangle myTri: ");
		myTri.print();
	}
}
```

Setting Visibility Levels

- You set the visibility level of every data field and class method individually by placing the visibility modifier in front of the declaration
- The default visibility level, if you don't specify any visibility modifiers, is package-private
  - This means that the data field or class method can be used only by other methods in the same *package*
- In this course, you should *always* set the visibility of every data field and class method to be explicitly either **`public`** or **`private`**

Aside: Packages

- Java provides *packages* to group together classes that are part of a related set of functionality
- A class is made part of a package by putting the `package` keyword and a package name at the top of the class file
  - » Generic form: `package PACKAGE_NAME;`
  - » For example: `package edu.wit.cs.comp1000.examples;`
- The default visibility level, package-private, provides access to other classes in the same package only
- Built-in classes are part of built-in packages such as `java.io`

Why `private`?

- By making data fields **`private`**, you ensure that they are not used outside of the class methods
- In this course, you should always make your data fields **`private`**
- Methods are made **`private`** if they are only used internally in the class, and should not be called elsewhere
- In other words, **`private`** methods are commonly used to hide the implementation details of a class
  - » In this course, most (not all) of your methods will be **`public`**

Example: `private` Methods

PlayingCard.java:

```
public class PlayingCard {
	private String mySuit;
	private int myRank;
	private void validate() {
		if (!mySuit.equals("Clubs") && !mySuit.equals("Diamonds") &&
		   !mySuit.equals("Spades") && !mySuit.equals("Hearts")) {
			System.out.println("Invalid Suit!");
			System.exit(0);
		}
		if (myRank < 1 || myRank > 13) {
			System.out.println("Invalid Rank!");
			System.exit(0);
		}
	}
	public void setCard(String suit, int rank) {
		mySuit = suit;
		myRank = rank;
		validate();
	}
	public void print() {
		if (myRank == 13) {
			System.out.print("King");
		} else if (myRank == 12) {
			System.out.print("Queen");
		} else if (myRank == 11) {
			System.out.print("Jack");
		} else if (myRank == 1) {
			System.out.print("Ace");
		} else {
			System.out.print(myRank);
		}
		System.out.println(" of " + mySuit);
	}
}
```

`private` method, can't be called from outside of the `PlayingCard` class

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		PlayingCard c = new PlayingCard();
		c.setCard("Spades", 12);
		c.print();

		c.setCard("sabers", 1);
	}
}
```

Better Printing with `toString()`

- By default, you can't print an object directly
  - » How would the JVM know what to print or how to format it?
  - » If you do print an object it will simply print the memory address of that object
- A `print()` method that prints out the object is ok, but a better solution is to have a `toString()` method that returns a `String` that represents the object
  - » That way, the caller can include that `String` directly in their own output statements
  - » In fact, Java will do the conversion automatically for you if you name the method `toString()`!

Example: toString()

PlayingCard.java:

```
public class PlayingCard {
	private String mySuit;
	private int myRank;
	private void validate() {
		if (!mySuit.equals("Clubs") && !mySuit.equals("Diamonds") &&
		   !mySuit.equals("Spades") && !mySuit.equals("Hearts")) {
			System.out.println("Invalid Suit!");
			System.exit(0);
		}
		if (myRank < 1 || myRank > 13) {
			System.out.println("Invalid Rank!");
			System.exit(0);
		}
	}
	public void setCard(String suit, int rank) {
		mySuit = suit;
		myRank = rank;
		validate();
	}
	public String toString() {
		String output = "";
		if (myRank == 13) {
			output += "King";
		} else if (myRank == 12) {
			output += "Queen";
		} else if (myRank == 11) {
			output += "Jack";
		} else if (myRank == 1) {
			output += "Ace";
		} else {
			output += myRank;
		}
		output += " of " + mySuit;
		return output;
	}
}
```

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		PlayingCard c = new PlayingCard();
		c.setCard("Spades", 12);

		System.out.println("My card is the " + c.toString());
		// even better:
		System.out.println("My card is the " + c);
	}
}
```

Formatting Strings

- It is often the case that you want to format the `String` you create in `toString()` in the same way as we do with `printf()`
  - » To control decimal places, etc
- The `String.format()` method creates `String` objects using the same syntax as `printf()`
- This is useful in many cases, not just in `toString()` methods
- Example:

```
String s = String.format("value=%.2f%n", val);
```

Example: `String.format()`

Rectangle.java:

```
public class Rectangle {
	private double length;
	private double width;

	public void setLength(double newLength) {
		length = newLength;
	}
	public void setWidth(double newWidth) {
		width = newWidth;
	}
	public double area() {
		return length * width;
	}
	public String toString() {
		String output = String.format("length: %.3f, ", length);
		output += String.format("width: %.3f, ", width);
		output += String.format("area: %.3f", area());
		return output;
	}
}
```

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		Rectangle rectA = new Rectangle();
		Rectangle rectB = new Rectangle();

		rectA.setLength(4.2);
		rectA.setWidth(10.0);
		rectB.setLength(3.8);
		rectB.setWidth(2.5);

		System.out.println("Rectangle A: " + rectA);
		System.out.println("Rectangle B: " + rectB);
	}
}
```

Exercise

- Modify your Triangle class by converting your `print()` method into a `toString()` method. Update `main()` accordingly.

Answer

Triangle.java:

```
public class Triangle {
	private double base;
	private double height;

	public void setBase(double newBase) {
		base = newBase;
	}

	public void setHeight(double newHeight) {
		height = newHeight;
	}

	public double area() {
		return 0.5 * base * height;
	}

	public String toString() {
		String output = String.format("base: %.3f, ", base);
		output += String.format("height: %.3f, ", height);
		output += String.format("area: %.3f", area());
		return output;
	}
}
```

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		Triangle myTri = new Triangle();

		myTri.setBase(17.348);
		myTri.setHeight(104.6);

		System.out.println("Triangle myTri: " + myTri);
	}
}
```

Objects as Arguments

- Objects can be used as method parameters, just like `int`, `double`, etc
  - » We've seen numerous examples of this
  - » Most recently when we passed `Scanner` and `PrintWriter` objects as arguments into methods
- Unlike primitive types, objects are passed into methods by *reference*
- This means that changes made to an object in a method are actually modifying the object in the calling method (just like arrays, which are actually objects!)

Example: Object as Arguments

Temperature.java:

```
public class Temperature {
	private double tempC;
	public void setCelsius(double tempCelsius) {
		tempC = tempCelsius;
	}
	public void setFahrenheit(double tempF) {
		tempC = (5.0 / 9.0) * (tempF - 32);
	}
	public double getCelsius() {
		return tempC;
	}
	public double getFahrenheit() {
		return ((9.0 / 5.0) * tempC) + 32;
	}
	public String toString() {
		String o = String.format("%.2f C", tempC);
		o += String.format(" (%.2f F)",
				getFahrenheit());
		return o;
	}
}
```

ClassExamples.java:

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class ClassExamples {

	public static void getTemperature(Scanner input, Temperature t) {
		System.out.print("Enter a temperature in Fahrenheit: ");
		try {
			t.setFahrenheit(input.nextDouble());
		} catch (InputMismatchException ex) {
			System.out.println("Invalid temperature!");
			System.exit(0);
		}
	}

	public static void main(String[] args) {
		Scanner keyboardInput = new Scanner(System.in);
		Temperature myTemp = new Temperature();

		getTemperature(keyboardInput, myTemp);

		System.out.println("That is " + myTemp);
	}

}
```

Objects in Class Methods

- You can also pass in objects as method arguments to class methods of the same class as the object
- It's easy to get confused in these situations because there are multiple objects of the same type
  - » The argument object and the object that was used to invoke the method in the first place
- Objects of the class type can be created and returned from within a method of that class as well
  - » For example, to return a new object of that type
  - » It's even easier to get confused in this situation

Example: Objects in Class Methods

Temperature.java:

```
public class Temperature {
	private double tempC;
	public void setTemperature(Temperature other) {
		tempC = other.tempC;
	}
	public Temperature plus(Temperature other) {
		Temperature result = new Temperature();
		result.tempC = tempC + other.tempC;
		return result;
	}
	public void setCelsius(double tempCelsius) {
		tempC = tempCelsius;
	}
	public void setFahrenheit(double tempF) {
		tempC = (5.0 / 9.0) * (tempF - 32);
	}
	public double getCelsius() {
		return tempC;
	}
	public double getFahrenheit() {
		return ((9.0 / 5.0) * tempC) + 32;
	}
	public String toString() {
		String o = String.format("%.2f C", tempC);
		o += String.format(" (%.2f F)",
				getFahrenheit());
		return o;
	}
}
```

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		Temperature t1 = new Temperature();
		Temperature t2 = new Temperature();
		Temperature t3 = new Temperature();

		t1.setCelsius(50);
		t2.setFahrenheit(50);
		t3.setTemperature(t2);

		Temperature t4 = t1.plus(t3);

		System.out.println(t1 + " + " + t3 + " = " + t4);
	}
}
```

The `this` keyword

- There is a special keyword that can be helpful when you need to refer to the "current" object within a class method
- The `this` keyword
- It is particularly useful to clarify which object you are accessing when there are multiple objects
- You will never be required to use it in this course, but if you find it useful then take advantage of it
  - » In other words, if it makes sense to you then use `this` and if not then don't

Example: `this`

Temperature.java:

```
public class Temperature {
	private double tempC;
	public void setTemperature(Temperature other) {
		this.tempC = other.tempC;
	}
	public Temperature plus(Temperature other) {
		Temperature result = new Temperature();
		result.tempC = this.tempC + other.tempC;
		return result;
	}
	public void setCelsius(double tempCelsius) {
		tempC = tempCelsius;
	}
	public void setFahrenheit(double tempF) {
		tempC = (5.0 / 9.0) * (tempF - 32);
	}
	public double getCelsius() {
		return tempC;
	}
	public double getFahrenheit() {
		return ((9.0 / 5.0) * tempC) + 32;
	}
	public String toString() {
		String o = String.format("%.2f C", tempC);
		o += String.format(" (%.2f F)",
				getFahrenheit());
		return o;
	}
}
```

Using `this` refers to the object that was used to invoke the method

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		Temperature t1 = new Temperature();
		Temperature t2 = new Temperature();
		Temperature t3 = new Temperature();

		t1.setCelsius(50);
		t2.setFahrenheit(50);
		t3.setTemperature(t2);

		Temperature t4 = t1.plus(t3);

		System.out.println(t1 + " + " + t3 + " = " + t4);
	}
}
```

In this invocation of `plus()`, `this` will refer to `t1`

Constructors

- Constructors are special class methods that are used for initialization, to *construct* an object
- A class can have multiple constructors that have different parameter lists, however any one object can only be initialized with one constructor
- The method name for a constructor is required to be the same as the name of the class
- Constructors have no return type (not even **`void`**)
- Except under very special circumstances, constructors should always be **`public`**

Example: Constructor

Stock constructor with two arguments

Stock.java:

```
public class Stock {
	private int shares;
	private double value;

	public Stock(int initialShares, double initialValue) {
		shares = initialShares;
		value = initialValue;
	}

	public String toString() {
		String output = String.format("#shares=%d, ", shares);
		output += String.format("value=%.3f", value);
		return output;
	}
}
```

Calling a Constructor

- Constructors are called automatically when you create an object with **new**
- They can not be called again after an object is created
- Only one constructor can be called per object, and one constructor is always called
- You specify the arguments to the constructor in the parentheses when you create the object
  - » Example:

```
Stock goog = new Stock(10, 716.8);
```

Example: Using the Constructor

Stock.java:

```
public class Stock {
	private int shares;
	private double value;

	public Stock(int initialShares, double initialValue) {
		shares = initialShares;
		value = initialValue;
	}

	public String toString() {
		String output = String.format("#shares=%d, ", shares);
		output += String.format("value=%.3f", value);
		return output;
	}
}
```

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		Stock goog = new Stock(10, 716.8);

		System.out.println("Google stock: " + goog);
	}
}
```

This calls the `Stock` constructor, just like any other method, and passes in `10` for `initialShares` and `716.8` for `initialValue`

Exercise

- Write a class named `Account` which has two (private) data fields: the balance of the account and the name of the account. Include a constructor that allows you to set the name and the initial balance of the account. Also include a `toString()` method that puts both the name and balance in the `String`. Write a `main()` method to test the class.

Answer

Account.java:

```
public class Account {
	private String name;
	private double balance;

	public Account(String accountName, double initialBalance) {
		name = accountName;
		balance = initialBalance;
	}

	public String toString() {
		String output = name;
		output += String.format(": $%.2f", balance);
		return output;
	}
}
```

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		Account checking = new Account("Checking", 0.93);
		System.out.println(checking);
	}
}
```

Multiple Constructors

- You can define as many constructors as you want for each class, so long as they conform to the normal method rules
- The parameter lists have to be different, meaning different types or different numbers of parameters
  - Method overloading!
- The correct constructor is automatically chosen based on the arguments provided

Example: Multiple Constructors

New constructor with a single `double` argument

Stock.java:

```
public class Stock {
	private int shares;
	private double value;

	public Stock(double initialValue) {
		shares = 0;
		value = initialValue;
	}
	public Stock(int initialShares, double initialValue) {
		shares = initialShares;
		value = initialValue;
	}

	public String toString() {
		String output = String.format("#shares=%d, ", shares);
		output += String.format("value=%.3f", value);
		return output;
	}
}
```

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		Stock goog = new Stock(10, 716.8);
		Stock msft = new Stock(52.84);
		System.out.println("Google stock: " + goog);
		System.out.println("Microsoft stock: " + msft);
	}
}
```

Calling the new constructor

Default Constructor

- One special constructor is the *default* constructor
- This is the constructor used when no arguments are provided when the object is created
  - » Example: `Stock csco = new Stock();`
- If you define no constructors for a class, the compiler automatically adds a default constructor that does nothing
  - » That's how all of our previous examples worked
- If you define *any* constructors for a class (not necessarily a default constructor), the compiler does *NOT* add a blank default constructor for you

Example: No Default Constructor

No default constructor is included, but there are other constructors

Stock.java:

```
public class Stock {
	private int shares;
	private double value;

	public Stock(double initialValue) {
		shares = 0;
		value = initialValue;
	}
	public Stock(int initialShares, double initialValue) {
		shares = initialShares;
		value = initialValue;
	}

	public String toString() {
		String output = String.format("#shares=%d, ", shares);
		output += String.format("value=%.3f", value);
		return output;
	}
}
```

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		Stock goog = new Stock(10, 716.8);
		Stock msft = new Stock(52.84);
		Stock csco = new Stock(); // build error
		System.out.println("Google stock: " + goog);
		System.out.println("Microsoft stock: " + msft);
		System.out.println("Cisco stock: " + csco);
	}
}
```

No default constructor means you can not create a `Stock` object with no arguments

Example: Default Constructor

Stock.java:

```
public class Stock {
	private int shares;
	private double value;

	public Stock() {
		shares = 0;
		value = 0;
	}
	public Stock(double initialValue) {
		shares = 0;
		value = initialValue;
	}
	public Stock(int initialShares, double initialValue) {
		shares = initialShares;
		value = initialValue;
	}

	public String toString() {
		String output = String.format("#shares=%d, ", shares);
		output += String.format("value=%.3f", value);
		return output;
	}
}
```

Default constructor (no arguments)

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		Stock goog = new Stock(10, 716.8);
		Stock msft = new Stock(52.84);
		Stock csco = new Stock();
		System.out.println("Google stock: " + goog);
		System.out.println("Microsoft stock: " + msft);
		System.out.println("Cisco stock: " + csco);
	}
}
```

Calling the default constructor

Example: Automatic Default Constructor

No constructors defined, so a default constructor that does nothing is automatically added

Default constructor is called, which does nothing and initializes no data fields

User.java:

```
public class User {
	private String username;
	private int id;

	public void setId(int newId) {
		id = newId;
	}
	public void setUsername(String newUsername) {
		username = newUsername;
	}
	public String toString() {
		return username + ": " + id;
	}
}
```

ClassExamples.j

```
public class ClassExamples {
	public static void main(String[] args) {
		User bbr = new User();
		System.out.println(bbr);
		bbr.setId(2716057);
		bbr.setUsername("bender");
		System.out.println(bbr);
	}
}
```

Exercise

- Modify your `Account` class to include a default constructor that sets the balance to $0.00 and the name to "`Account`". Also add a method named `adjust()` that allows you to adjust the balance by a positive or negative amount. Test the new methods in `main()`.

Answer

Account.java:

```
public class Account {
	private String name;
	private double balance;

	public Account() {
		name = "Account";
		balance = 0;
	}
	public Account(String accountName, double initialBalance) {
		name = accountName;
		balance = initialBalance;
	}

	public void adjust(double amount) {
		balance = balance + amount;
	}
	public String toString() {
		String output = name;
		output += String.format(": $%.2f", balance);
		return output;
	}
}
```

ClassExamples.java:

```
public class ClassExamples {
	public static void main(String[] args) {
		Account checking = new Account("Checking", 0.93);
		System.out.println(checking);
		Account account = new Account();
		account.adjust(1000);
		account.adjust(-250);
		System.out.println(account);
	}
}
```

One Last Detail: `static` vs Instance variables

- All of the data fields we've defined in our classes so far have been *instance* variables
- This means that each object (*instance* of a class) has a separate variable in memory for each data field
  - » For example, every `Account` object has its own `name` variable and its own balance `balance`
- The alternative is a `static` data field, where every object of the class *shares* a single variable
  - » Only one variable in memory for all instances of the class
  - » If one object changes a `static` data field, all objects of that class will be affected

Example: A `static` Data Field

User.java:

```
public class User {
	private String username;
	private int id;
	private static int numberOfUsers = 0;

	public User(String newUsername) {
		username = newUsername;
		id = numberOfUsers;
		numberOfUsers++;
	}

	public String toString() {
		return username + ": " + id;
	}
}
```

ClassExamples.java:

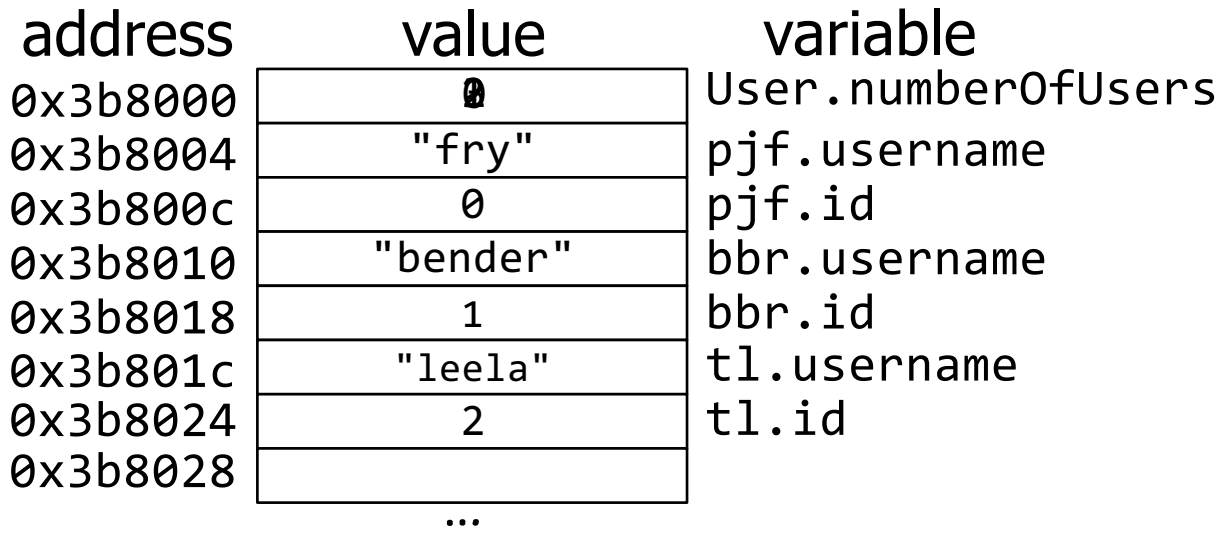
```
public class ClassExamples {
	public static void main(String[] args) {
		User pjf = new User("fry");
		User bbr = new User("bender");
		User tl = new User("leela");
		User jaz = new User("zoidberg");

		System.out.println(pjf);
		System.out.println(bbr);
		System.out.println(tl);
		System.out.println(jaz);
	}
}
```

`static` Data Fields in Memory

- `static` data fields aren't tied to any one object like instance data fields

```
User pjf = new User("fry");
User bbr = new User("bender");
User tl = new User("leela");
```



`static` Class Methods

- Class methods can also be **`static`**
  - » Like `main()` or other methods we've made earlier in the semester
- The meaning is similar: a **`static`** method is shared by all instances of the class
- A **`static`** method can NOT use or modify any instance (non-**`static`**) data fields directly
  - » If you have a method that does not use or modify any instance data fields, then make it **`static`**
- A **`static`** method can be called from other methods with the class name (not any particular object)

Example: `static` Class Methods

User.java:

```
public class User {
	private String username;
	private int id;
	private static int numberOfUsers = 0;

	public User(String newUsername) {
		username = newUsername;
		id = numberOfUsers;
		numberOfUsers++;
	}

	public static int getNumberOfUsers() {
		retur numberOfUsers;
	}

	public String ng() {
		return ": " + id;
	}
}
```

Class

```
public class ClassExa
	public static voi ng[] args) {
		User pjf = n fry");
		User bbr = new bender");
		User tl = new Use eela");
		User jaz = new User zoidberg");

		int totalUsers = User.getNumberOfUsers();
		System.out.println("Total number of users: " + totalUsers);

		System.out.println(pjf);
		System.out.println(bbr);
		System.out.println(tl);
		System.out.println(jaz);
	}
}
```

Calling the `static` method is done with the class name (`User`) then a dot then the method name (`getNumberofUsers()`)

This method is `static`, and therefore can't use the `username` or `id` data fields

Wrap Up

- A class defines a complex variable type
  - » Contains its own data fields and methods that are only for use with objects of that class
- There are many predefined classes in Java that we've been using all semester including `String`, `Scanner`, and `PrintWriter`
- You can also define your own classes
  - » Often done to represent an entity in your program that requires more than one variable
- This is just the beginning of object oriented (OO) software development