CSCI 136
Data Structures &
Advanced Programming

Williams College

Fall 2020

Instructors: The Bills (J & L)

Today' s Outline

- Why is 136 taught in Java?

- Object Oriented Programming (OOP)!
  - OOP as a (powerful) way to organize your code
  - Discuss select Java features that support OOP
    - Classes & Objects
    - Access Modifiers
    - Interfaces
    - `static` (variables and functions)

WHY JAVA?

Java is a compiled language

- Java code is sent to a compiler that statically verifies the code follows the language's rules

```
$ javac HelloWorld.java
$ ls
HelloWorld.java
HelloWorld.class
```

- The resulting `.class` file can then be run by the Java Virtual Machine (JVM)

```
$ java HelloWorld
Hello World!
```

- Question: Why is this good?

Java is a compiled language

- Why is this good? (many reasons…)
  - We can detect certain errors before they happen
    - Can then ask the compiler for more information (or to run again with different settings)
    - Compile-time errors vs. Run-time errors
  - Efficient representation of code
    - Compiler can apply many complex optimizations without much additional work from programmers
    - Compiler does work once, but program may be run many times

Java is Object-Oriented

- Language often influences the way we approach/think about a problem

- Object-oriented programming is how we will design our programs in this course
    - OOP may seem unnatural at first, but try to think in the OOP mindset and give it a chance; it’ll help to build intuition for its benefits and limits

OOP: OBJECT ORIENTED PROGRAMMING

Classes, objects, and interfaces

- Classes let us define our own types.
- Objects are instances of class types
- *Example*: Think about the abstract concept of a car. Here are three instances of a car:

- Conceptually, all these cars have the same high-level interface (wheels, doors, color, transmission, top speed, etc.) but individual cars differ in their details
  - In OOP paradigm, we could *define* a car class, and then *instantiate* that class to create individual car objects.

Object-Oriented Programming

- Objects are building blocks of Java software
- Programs are collections of interacting objects
  - Cooperate to complete tasks
  - Represent the "state" of the program
  - Communicate by sending messages to each other
    - Through *method invocation*

Object-Oriented Programming

- With enough creativity, objects can model almost anything:
  - Physical items – cars, dice, book
  - Concepts – time, relationships
  - Processing – sort, simulation, gameplay
- Objects contain:
  - State (instance variables)
  - Functionality (methods)

Object Support in Java

- Java supports the creation of programmer-defined types called *class types*
- A *class declaration* defines data components and functionality of a type of object
  - Data components: *instance variable declarations*
  - Functionality: *method declarations*
    - *Constructor(s)*: special method(s) that describe the steps needed to create an object (*instance*) of this class type

A Simple Class

Task: Define a type that stores information about a student: name, age, and a single grade.

- Declare a Java class called `Student` with data components (*fields/instance variables*):

```
String name;
int age;
char grade;
```

- and methods for accessing/modifying fields:
  - "Getters": `getName`, `getAge`, `getGrade`
  - "Setters": `setAge`, `setGrade`
- Declare a constructor, also called `Student`

```
class Student {
      // instance variables
      int age;
      String name;
      char grade;

      // A constructor
      Student(int theAge, String theName,
                  char theGrade) {
            age = theAge;
            name = theName;
            grade = theGrade;
      }

      // Methods for accessing/modifying objects
      // ...see next slide...
      int getAge() { return age; }

      String getName() { return name; }

      char getGrade() { return grade; }

      void setAge(int theAge) {
            age = theAge;
      }

      void setGrade(char theGrade) {
            grade = theGrade;
      }
} // end of class declaration from previous slide
```

Constructors

Principle: Use constructors to initialize the ***state*** *of an object, nothing more.*

  - What is state? instance variables

- Frequently constructors are short simple methods
- More complex constructors will typically use helper methods. Why?
  - A class may have more than one constructor!
  - Your constructors can call other constructors or helper methods in order to reuse code
    - **Never copy/paste code!!!**

IMPROVING THE STUDENT CLASS

Access Modifiers

- **`public`**, **`private`**, and **`protected`** are called *access modifiers*
  - They control access of other classes to instance variables and methods of a given class
    - `public` : Accessible to all other classes
    - `private` : Accessible only to the class declaring it
    - `protected` : Accessible to the class declaring it and its subclasses
- Data-Hiding Principle (encapsulation)
  - Make instance variables `private/protected`
  - Use `public` methods to access/modify object data

```java
public class Student {
      // instance variables
      protected int age;
      protected String name;
      protected char grade;

      // A constructor
      public Student(int theAge, String theName,
                  char theGrade) {
            age = theAge;
            name = theName;
            grade = theGrade;
      }

      // Methods for accessing/modifying objects
      // ...see next slide...
      public int getAge() { return age; }

      public String getName() { return name; }

      public char getGrade() { return grade; }

      public void setAge(int theAge) {
            age = theAge;
      }

      public void setGrade(char theGrade) {
            grade = theGrade;
      }
} // end of class declaration from previous slide
```

TESTING THE STUDENT CLASS

Testing the Student Class

```java
public class TestStudent {

   public static void main(String[] args) {
      Student a = new Student(18, ”Bill J", 'B');
      Student b = new Student(19, ”Bill L", 'A');
      // Some code to nicely print student details
      System.out.println(a.getName() + ", " +
         a.getAge() + ", " + a.getGrade());
      System.out.println(b.getName() + ", " +
         b.getAge() + ", " + b.getGrade());
      // Ugly printing (calls default toString())
      System.out.println(a);
      System.out.println(b);
   }
}
```

“Special” Methods

- Everything “inherits” from the class `java.lang.Object`
- In particular, we’ll take advantage of a few methods repeatedly in this course:
    - `String toString()`
    - `boolean equals(Object other)`
    - `int hashCode()`
- Today, let’s just look at `toString()`

Worth Noting

- We can create as many Student objects as we need, including arrays of Students

```
Student[] class = new Student[3];
class[0] = new Student(18, ”Huey", 'A');
class[1] = new Student(20, ”Dewey", 'B');
class[2] = new Student(21, ”Louie", 'A');
```

- Fields are *private*: only accessible in Student class
- Methods are *public*: accessible to other classes
- Some methods return values, others do not
    - `public String getName();`
    - `public void setAge(int theAge);`

More Gotchas

```
public class Student {
      // instance variables
      private int age;
      private String name;
      private char grade;

      // A constructor
      public Student(int age, String name,
                  char grade) {
            // What would age, name, grade
            // refer to here...?
      }
```

For clarity, can use ‘this’

```
public class Student {
      // instance variables
      private int age;
      private String name;
      private char grade;

      // A constructor
      public Student(int age, String name,
                  char grade) {
            this.age = age;
            this.name = name;
            this.grade = grade;
      }
```

INTERFACES: A WAY TO STANDARDIZE BEHAVIOR

Interfaces

- We’ve used the term interface to colloquially describe the way that we interact with objects, but a Java `interface` is a contract
  - Defines methods (name, parameters, return types) that a class *must* implement
  - Kind of like a “class recipe”
- Multiple classes can *implement* the same interface, and we are guaranteed that they all implement the required methods

A Student Interface

Task: Rework the Student class into an interface that defines the behaviors that any "student class type" must provide in order to be useful. Note, we only care about *behavior*, not *implementation*.

Interfaces do not specify state or provide code*.

Declare a Java interface called `Student` with public methods:

- Getters: `getName`, `getAge`, `getGrade`
- Setters: `setAge`, `setGrade`

Student Interface

```
public interface Student {
      // Note: no instance variables, constructor,
      //       or implementation
      public int getAge();
      public String getName();
      public char getGrade();
      public void setAge(int theAge);
      public void setGrade(char theGrade);
}
```

Interfaces

- A class can *implement* an interface by providing code for each required method.
- If we have code that depends only on the functionality described in the interface, that code can work for objects of any class that implements the interface!
  - Recall our eternal goal: write code exactly once

A Williams Student

Task: Write a `WilliamsStudent` class that implements the `Student` interface. Note, it must implement *everything* in the interface, but it can also add extra functionality.

- `protected String[] clubs;`
- `public String[] getClubs();`
- `public void setClubs(String clubs[]);`

(Note: I'm told that every Williams student participates in at least fourteen extra-curricular activities)

(NO) STATIC

Static Variables

- Variables can either be “attached” to the class or to instances of the class.
  - Static variables **are not** associated with any one object’s state. They are usually properties or definitions.
  - Non-static variables are called instance variables because they are tied to exactly one instance of an object. They can be accessed with the keyword ‘this’.
  - Ask yourself: Is it possible that the value of this variable will vary across different objects?
    - Consider a `Rectangle` class :
      - `numSides;` static (all rectangles have 4 sides)
      - `height;` not static (rectangles can have different dimensions)

Static Methods

- Methods can either be "attached" to the class or to instances of the class.
  - Static methods **do not** depend on the state of the object. They can be answered without anything that could reference the keyword "this". Called using the class name.
  - Non-static methods rely on an object's state, often depending on the values of instance variables. Called on an instance.
  - Ask yourself: Does this method depend on the state of the object, or is it always the same regardless?
    - Consider a `Rectangle` class:
      - `getArea();` not static (depends on a particular rectangle's dims)
      - `calculateArea(int h, int w);` static (formula; all info provided as inputs)

More Gotchas

```
public static void main(String[] args) {
     // try to access a student’s age
     System.out.println(getAge());
     // Wrong! Which student? getAge is not static,
     // so we need to call it on a particular object

     // try to access a student’s age (correctly)
     Student s = new WilliamsStudent(18, “Ron”, ‘C’);
     System.out.println(s.getAge());
}
```