Classes and Objects

Part II: Class design, including:
Overloading
this
Aggregation
static
Packages
enum
null

Class Design Review

- A class is a *concept* of an object
- Classes we write include:
  - Private instance variables
  - constructors
  - getters and setters
  - toString
  - private and public class-specific methods
- Once we define a class, we can instantiate as many objects of that class as we need
  - Using the new operator and the constructor
- We then invoke methods on those objects

METHOD OVERLOADING

Method Overloading

- *Method overloading* allows multiple methods with the same name.
- Overloaded methods differ by:
  - The number of parameters,
  - The type of parameters, or
  - The order of the parameters

Method Overloading

- The `println` method is overloaded:

```
public void println  (String s)
public void println  (int i)
public void println  (double d)
```

- The following lines actually invoke different versions of the `println` method:

```
System.out.println("Hello");
System.out.println(35);
System.out.println(42.5);
```

Method Overloading

- If a method is overloaded, the name alone is not sufficient to determine which method is being called. The compiler determines which method to invoke by analyzing the parameters.
- The return type is *not* part of the signature
  - Overloaded methods **cannot** differ only by return type

Method Overloading (continued)

```
double tryMe(int x) {
   return x + 0.375;
}

int tryMe(int x) {
   return (int) x;
}

double tryMe(int x, double y) {
   return x*y;
}

double tryMe(int a, double b) {
   return a*b;
}
```

Compiler Error!!!

When to Use Method Overloading

- Use method overloading only when you have multiple methods that take different parameters but do the *same* thing.
  - You often want the client to think it's the same method (e.g., println).
- You should **not** overload methods if two methods perform different tasks.

Practice

- Overload the roll method.

Overloading Constructors

- It is common to overload constructors
- This provides multiple ways to initialize a new object
  - This is commonly used when you have default values for instance data that can be assigned when the user does not supply values

Practice

- Overload the constructor of the Die class.
- Overload the constructor of the AudioItem class.
- Overload the constructor of the Employee class.

The `this` Reference

- The `this` reference allows an object to refer to itself
- `this` refers to the current object
- When used inside a method, `this` refers to the object through which the method is invoked

The `this` Reference

- The `this` reference can be used to distinguish between instance variables and formal parameters with the same name, often used in constructors

```
public Account (String name, double balance) {
   this.name = name;
   this.balance = balance;
}
```

this.X is the instance data variable

X is the parameter

The `this` Reference

- The second use of the `this` keyword is to call an overloaded constructor
  - Note: this statement *must* be the first line in the constructor
- It's common (and good practice) to invoke the "longest" constructor (the one with the most parameters) from the other constructors-passing the parameters and default values.
  - Use `this` to do so!

Practice

- Modify the Die class.
  - Change the parameter names in this class to the same as the instance variable names.
  - Use `this` to invoke the overloaded constructor.
- Modify the AudioItem class.
  - Change the parameter names in this class to the same as the instance variable names.
  - Use `this` to invoke the overloaded constructor.

AGGREGATION

Class Relationships

- Dependency: A *uses* B
- Aggregation or Composition: A *has-a* B
- Inheritance: A *is-a* B

Class Dependency (“Uses”)

- A *dependency* exists when one class relies on another class in some way
  - Examples: creating an object of that class, invoking the methods of that class, etc.

```
public class Driver {
  private static void main(String[] args){
        AudioItem i1 = new AudioItem(…);
```

  - Here our `Driver` class *depends on* the `AudioItem` class
    - `Driver` uses `AudioItem`

Class Aggregation or Composition ("Has A")

- An *aggregate* is an object that is made up of other objects
  - An *instance* of one object is part of what defines another object
  - Example: A car *has-a* steering wheel
- An aggregate object contains references to other objects *as instance data*.
  - Example: `Employee` *has one or more* `String`
  - Example: `DicePair` *has one or more* `Die`

Aggregation

- Aggregation helps support code reuse and maintenance.
- If a class already exists to do something, use it, rather than rewrite it!

Practice

- Write an AudioStoreAccount class.
  - name, account ID, balance, list of audio items owned
  - This is aggregation! An AudioStoreAccount *has* AudioItem objects as part of what describes it.
- Update the driver program to create some accounts.

STATIC

Variables Revisited

- Instance data
  - Declared in the class
  - Used anywhere in the class
  - Lives as long as the object lives
  - One version for each object
- Local data
  - Declared inside of a method
  - Used only in that method
  - Dies when the method ends

Instance Variables

- For instance variables, each object has its own data space

```
private String firstName;
```

  - Each Student has it’s own first name.
- You update instance data through public methods invoked on an object.

```
student1.setFirstName("Jim");
```

  - Change the `firstName` of the object `student1`

Static Variables

- Static variables (also called class variables) are associated with the class itself, not with any single instance of the class
- One copy/version for the whole class!

Static Variables

- For static variables, only **one** copy of the variable exists for *all* objects of that class

```
public static int numberOfStudents;
```

  - There is only one count of the number of students and it is shared by all objects of the Student class.
  - If `student1` updates `numberOfStudents`, it's changed for `student2` as well, because it's the *same variable!*

Static Variables

- You reference static variables through the name of the class, not through any particular object.
  - `Student.getNumberOfStudents();`

Static Variables

- Changing the value of a static changes it for all objects of that class

```
public Student(…)   {
 …
 Student.numberOfStudents++;
}
```

- Memory space for static variables is created when the class is first referenced.

Invoking Methods Revisited

- Most methods are invoked through an instance of a class:
  - We create an instance with the new operator
  - We invoke a method with the dot operator
- Examples:
  - `Scanner scan = new Scanner(System.in);`
    `scan.nextLine()`
  - `Die d1 = new Die();`
    `d1.roll();`

Static Methods

- Static methods (also called class methods) are invoked **not** through an object but through the **class** name
  - `double answer = Math.sqrt (25)`
  - `double number = Math.random();`
- Static methods are more like functions associated with the class
  - They should not be used if a method represents an object's functionality.
  - They **cannot** be used if they require access to instance data variables.

Static Methods

- Static methods are invoked through the class, not through any object.

```
public static int numberOfStudents;

public static int getNumberOfStudents() {
        return numberOfStudents;
}
public static void setNumberOfStudents(int n){
        numberOfStudents = n;
}
```

Static Methods and Variables

- We declare static methods and variables with the `static` keyword
- A static method or variable is associated with the *class itself*, rather than with any individual instantiated object of the class
  - One copy/version for the whole class!

- By convention, visibility modifiers come first
  - `public static` not `static public`

Static Methods (cont.)

- Static methods:
  - *cannot* reference instance variables
    - Those variables don't exist until an object exists (and then each object has its own version of them)
  - *can* reference static variables and local variables

- Static methods:
  - *cannot* directly reference other non-static methods
    - Those must be referenced through an object
  - *can* reference other static methods

- You will get a compiler error if you try to do these things!

Accessing Variables and Methods

	Static Variables	Instance Variables
Static Methods	can access	cannot access
Instance Methods	can access	can access

Accessing Variables and Methods (continued)

	Static Variables	Instance Variables
Static Methods	can access	cannot access
Instance Methods	can access	can access

```
public Student () {
    …
    Student.numStudents++;
}
```

Accessing Variables and Methods (continued)

	Static Variables	Instance Variables
Static Methods	can access	cannot access
Instance Methods	can access	can access

```
public String getFirstName() {
   return firstName;
}
```

Accessing Variables and Methods (continued)

	Static Variables	Instance Variables
Static Methods	can access	cannot access
Instance Methods	can access	can access

```
public static int getNumStudents() {
   return Student.numStudents;
}
```

Accessing Variables and Methods (continued)

	Static Variables	Instance Variables
Static Methods	can access	cannot access
Instance Methods	can access	can access

```
public static int getStudentName() {
   return firstName;
}
```

would be invoked:

```
Student.getStudentName();
```

static methods are invoked through the class, not through an object… so which student's name should be returned??

Using Static Variables

- Shared data (be careful!)
  - Examples: a count of objects, a total across all objects
- Shared constants
  - Example: `MAX_VALUE`
  - Examples:
    - `BorderLayout.CENTER`
    - `JOptionPane.YES`
    - `Integer.MAX_VALUE`

Using Static Methods

- Utility or helper functions
  - send input, get a result
  - Example: Math.sqrt
- Accessing static variables or shared information
  - Example: getNumberOfstudents()

Practice

- Modify the AudioItem class to keep track of how many audio items exist.
- Modify the AudioStoreAccount class to keep track of how much money has been spent on AudioItems across *all accounts*.

PACKAGES

Packages

- Packages are used to group common classes together.
- Classes libraries are organized into packages
  - `java.util` package contains utility classes
    - `Scanner`
    - `Random`

- By convention, packages are lowercase

Packages

- To use a class that is in the **same package**, you can just refer to the class name directly.
- To use a class in a different package, you typically:
  - import a single class

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

  - import all classes in a package

```
import java.util.*;
// imports Scanner, Random, and more
```

Packages

- As a general rule of thumb, if you are using more than one class in a package, you can import the entire package.
- If you don’t import it and don’t use a fully qualified name, you will get a compiler error.
  - Java doesn’t know where the Scanner class is if you don’t tell it!

The `java.lang` Package

- All classes of the `java.lang` package are imported automatically into all programs.
- It's as if all programs contain:

```
import java.lang.*;
```

- This is why we don't have to import anything to use `System` or `String`

Class Libraries

- A *class library* is a collection of classes
- The *Java standard class library* is often called the Java API (Application Programming Interface)
- We rely heavily on many classes, but they are part of the API, not part of the language.
  - System
  - Scanner
  - String

Using Your Own Packages

- You can organize your own code into packages.
- Put a package statement at the top of your class.

```
package company;
```

- Your directory structure must match your package structure.
  - Employee class is in a file called Employee.java
  - company package is in a folder called company
  - All classes in the company package go inside the company folder

ENUMS

enum

- A way to provide a restricted set of values
- You can declare variables of this type.
- Examples
  - Sizes: Small, medium, large
  - Suits: Diamonds, hearts, spades, clubs
  - Semesters: Fall, summer, spring

```
enum Size {SMALL, MEDIUM, LARGE};

Size s1 = Size.LARGE;
// s1 can only hold the values SMALL, MEDIUM,
LARGE, or null

Size s2 = Size.SMALL;
```

Can’t we just use constants?

```
public static final int SMALL = 0;
public static final int MEDIUM = 1;
public static final int LARGE = 2;

public int size = SMALL;
```

- No type safety
  - `public setSize(int size) {`

  // someone could send in -9!
- Allows for illogical results
  - `public static final int FALL = 2;`
  - `FALL == LARGE`. Huh?!
- No easy way to translate to String output
- No way to iterate over all of the choices

Constants vs. enums

- Constants are good things! You should use them in your code.
  - Constants are good for single values like min, max, default values, etc.
- enums are good things! You should use them in your code.
  - enums are best when something has a predefined, finite set of possible values.

enums are Classes!

- You can add constructors, methods, and fields.
  - Constructors are invoked when the enum constants are constructed.
  - Methods and fields are used when you want to associate data or behavior with a constant
- All enums are subclasses of `Enum`.

Example

```
enum Size {
      SMALL("S"), MEDIUM("M"), LARGE("L"),
      EXTRA_LARGE("XL");

      private String abbreviation;

      private Size(String abbreviation) {
            this.abbreviation = abbreviation;
      }
      public String getAbbreviation() {
            return abbreviation;
      }
}
```

Methods for enums

- `toString`
  - Returns the name of the constant
- `static values()` method returns an array of all possible values
  - Example: `Size[] values = Size.values();`
- `ordinal` method returns the position of the constant in the declaration.
  - Starting from 0.
  - Example: `Size.LARGE.ordinal()` returns 2

Practice

- Add an enum to the Employee class to represent whether the employee is full time part time, or inactive.
  - Add data to the enum that represents whether that type of employee gets benefits.
  - Include a toString method.

Practice

- Write a class to represent a GradeRecord, described by:
  - Student name
  - numeric grade
  - letter grade (A, B, C, D, F)
- Write a driver program to create a Gradebook.
  - Have the user enter grades.
  - Print the number of passing grades and the number of As in the gradebook.

Practice

- Write classes to represent a Donation and a Donor. Write a driver program.
  - Donations are described by an amount and date.
  - Donors are described by name, phone number, and a list of donations.

NULL

null

- null is a keyword/reserved word in Java
- null represents **no value**

null as a Default Value

- Instance data variables are given a default value when they are declared.
  - Local variables are not.
- Each data type has a value that is used for the default value.
- null is the default value for objects when they have been declared but not initialized.
  - private int age; // default value for int is 0
  - private String name; // default value of name is null
  - private Student s; // default value of s is null

null as a Value

- null is a value that can be assigned to any *object* reference (variable).
- null cannot be assigned to primitives.
- Examples:
  - Student s = null; // allowed
  - int n = null; // not allowed
  - Integer m = null; // allowed

What can you do with null?

- Compare it with ==
- Examples:
  - if(student!=null) { … }
  - if(student!=null && student.meetsCriteria()) { … }

What **can't** you do with null?

- Pretty much anything else!
- Most importantly: you cannot invoke any methods or try to access any variables on null.
- Invoking a method on a null object will throw a NullPointerException, which will crash your program.
  - This is a bad exception because it is almost always a result of programmer error.
  - This is almost always entirely preventable.

What **can't** you do with null?

- Example:
  - String s = null
  - s.equals("hello"); // crash!
  - s=="hello"; // allowed- will not crash, returns false
- Example:
  - Student student = null;
  - System.out.println(student.name); // crash!

null and Linked Nodes

- null will matter a lot when we cover nodes and linked lists!!

null and Strings

- A string whose value is the empty String is **not** null.
- An empty String is a String that does not contain any characters. (But it is not null!)
  - String s1 = “”; // empty string! length = 0
  - String s2 = null; // no length- no value!
  - s1==s2; // false

null and Strings

```
String s1 = "";
String s2 = null;
String s3; // value is also null

System.out.println(s1.toUpperCase());
// allowed- but nothing will be printed!

System.out.println(s1.length());
// allowed and will print 0

System.out.println(s2.toUpperCase());
// not allowed- will crash with NullPointerException

System.out.println(s3.length());
// not allowed- will crash with NullPointerException
```