Java classes

Savitch, ch 5

Assignment 1: line segments

In Assignment 1 you were asked to implement methods for reading/querying a collection of line segments:

```java
int[][] readLineSegments(String fileName)
boolean areEqual(int i, int j, int[][] lineSegments)
boolean shareEndpoint(int i, int j, int[][] lineSegments)
boolean areIntersecting(int i, int j, int[][] lineSegments)
```

Why do we need to pass around the line-segments array?
Can we come up with a nicer representation of line-segments?

Line segments

- A better design:
  - A class that represents line-segments
    Stores the coordinates of a line segment and knows how to compare them.
  - A class that reads/stores the collection of line segments

- We will work towards this design in the next two assignments

Objects and classes

- object: An entity that combines state and behavior.
  - object-oriented programming (OOP): Writing programs that perform most of their behavior as interactions between objects.
- class: 1. A program/module. or, 2. A blueprint/template for an object.
- classes you may have used so far: String, Scanner, File
- We will write classes to define new types of objects.

Abstraction

- abstraction: A distancing between ideas and details.
  - Objects in Java provide abstraction:
    We can use them without knowing how they work.

- You use abstraction every day.
  - Example: Your portable music player.
    - You understand its external behavior (buttons, screen, etc.)
    - You don’t understand its inner details (and you don’t need to).

Class = blueprint, Object = instance

Music player blueprint

state:
- current song
- volume
- battery life
behavior:
- power on/off
- change station/song
- change volume
- choose random song

Music player #1

state:
- song = “Thriller”
- volume = 17
- battery life = 2.5 hrs
behavior:
- power on/off
- change station/song
- change volume
- choose random song

Music player #2

state:
- song = “Feels like rain”
- volume = 9
- battery life = 3.41 hrs
behavior:
- power on/off
- change station/song
- change volume
- choose random song

Music player #3

state:
- song = “Code Monkey”
- volume = 24
- battery life = 1.8 hrs
behavior:
- power on/off
- change station/song
- change volume
- choose random song
How often would you expect to get snake eyes?

If you're unsure on how to compute the probability then you write a program that simulates the process.

```java
public class SnakeEyes {
    public static void main(String[] args) {
        int ROLLS = 100000;
        int count = 0;
        Die die1 = new Die();
        Die die2 = new Die();
        for (int i = 0; i < ROLLS; i++) {
            if (die1.roll() == 1 && die2.roll() == 1) {
                count++;
            }
        }
        System.out.println("snake eyes probability: \(\) + (float) count / ROLLS); 
    }
}
```

Die object

- **State (data) of a Die object:**

<table>
<thead>
<tr>
<th>Instance variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numFaces</td>
<td>the number of faces for a die</td>
</tr>
<tr>
<td>faceValue</td>
<td>the current value produced by rolling the die</td>
</tr>
</tbody>
</table>

- **Behavior (methods) of a Die object:**

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>roll()</td>
<td>roll the die (and return the value rolled)</td>
</tr>
<tr>
<td>getFaceValue()</td>
<td>retrieve the value of the last roll</td>
</tr>
</tbody>
</table>

Object state:

instance variables

The Die class

- **The class (blueprint) knows how to create objects.**

Die class

- **The following code creates a new class named Die.**

```java
public class Die {
    int numFaces;
    int faceValue;
}
```

- Save this code into a file named Die.java.
- Each Die object contains two pieces of data:
  - an int named numFaces,
  - an int named faceValue
- No behavior (yet).
Instance variables

- **instance variable**: A variable inside an object that holds part of its state.
  - Each object has its own copy.
- Declaring an instance variable:
  ```java
  <type> <name>;
  ```
  ```java
  public class Die {
      int numFaces;
      int faceValue;
  }
  ```

Each Die object maintains its own numFaces and faceValue variable, and thus its own state

- **Declaring an instance variable**

```java
Die die1 = new Die();
Die die2 = new Die();
```

```java
numFaces: 5
faceValue: 2
```

```java
numFaces: 6
faceValue: 3
```

Accessing instance variables

- Code in other classes can access your object’s instance variables.
  - Accessing an instance variable: **dot operator**
    ```java
    <variable name>.<instance variable>
    ```
  - Modifying an instance variable:
    ```java
    <variable name>.<instance variable> = <value>;
    ```
  - Examples:
    ```java
    System.out.println("you rolled " + die.faceValue);
    die.faceValue = 20;
    ```

Client code

- **Die.java** can be made executable by giving it a main...
  - We will almost always do this... WHY?
  - To test the class Die before it is used by other classes
  - or can be used by other programs stored in separate .java files.
  - **client code**: Code that uses a class

```java
public class Die {
    int numFaces;
    int faceValue;
}
```

```java
main(String[] args) {  
    Die die1 = new Die();
    die1.numFaces = 6;
    die1.faceValue = 5;
    Die die2 = new Die();
    die2.numFaces = 10;
    die2.faceValue = 3;
    ...
}
```

Object behavior: methods

- Classes combine **state** and **behavior**.
- **instance variables**: define state
  - **instance methods**: define behavior for each object of a class. Are the way objects communicate with each other and with users.
- **instance method declaration**, general syntax:
  ```java
  public <type> <name> ( <parameter(s)> ) {
      <statement(s>)
  }
  ```

```java
public class Die {  
    int numFaces;
    int faceValue;
}
```
Rolling the dice: instance methods

```java
public class Die {
    int numFaces;
    int faceValue;
    public int roll (){
        faceValue = (int)(Math.random() * numFaces) + 1;
        return faceValue;
    }
}
```

Die die1 = new Die();
int value1 = die1.roll();

Die die2 = new Die();
int value2 = die2.roll();

Think of each Die object as having its own copy of the roll method, which operates on that object's state.

Object initialization: constructors

Die constructor

```java
public class Die {
    int numFaces;
    int faceValue;
    public Die (int faces) {
        numFaces = faces;
        faceValue = 1;
    }
    public int roll (){
        faceValue = (int)(Math.random()*numFaces) + 1;
        return faceValue;
    }
}
```

Die die1 = new Die(6);
Die die2 = new Die();

Constructors

- **constructor**: creates and initializes a new object
  - public <type> { <parameter(s)> } {<statement(s)> ;
  - For a constructor the <type> is the name of the class
  - A constructor runs when the client uses the new keyword.
  - A constructor implicitly returns the newly created and initialized object.
  - If a class has no constructor, Java gives it a default constructor with no parameters that sets all the object's fields to 0 or null.
  - we did this in Recap.java

Multiple constructors are possible

```java
public class Die {
    int numFaces;
    int faceValue;
    public Die () {
        numFaces = 6;
        faceValue = 1;
    }
    public Die () {
        numFaces = 10;
        faceValue = 1;
    }
}
```

Initializing objects

- When we create a new object, we can assign values to all, or some of, its instance variables:
  - Die die1 = new Die(6);
  - How do make that happen?
The Student class

Let's write a class called Student with the following state and behavior:

**Student**

- **State:**
  - String name
  - String id
  - int[] grades

- **Behavior:**
  - Constructor - takes id and name
  - numGrades - returns the number of grades
  - addGrade - adds a grade
  - getAverage - computes the average grade

Encapsulation

- **encapsulation:**
  - Hiding implementation details of an object from clients.
  - Encapsulation provides abstraction; we can use objects without knowing how they work.

  - an external view (its behavior)
  - an internal view (the state and methods that accomplish the behavior)

Implementing encapsulation

- Instance variables can be declared private to indicate that no code outside their own class can access or change them.
  - Declaring a private instance variable:
    ```
    private <type> <name>;
    ```
  - Examples:
    ```
    private int faceValue;
    private String name;
    ```

  - Once instance variables are private, client code cannot access them:
    ```java
    Roll.java:11: faceValue has private access in Die
    System.out.println("faceValue is " + die.faceValue);
    ^
    ```

Accessors and mutators

- We provide accessor methods to examine their values:
  ```java
  public int getFaceValue() {
    return faceValue;
  }
  ```

  - This gives clients read-only access to the object's fields.
  - Client code will look like this:
    ```java
    System.out.println("faceValue is " + die.getFaceValue());
    ```

- If required, we can also provide mutator methods:
  ```java
  public void setFaceValue(int value) {
    faceValue = value;
  }
  ```

  - Often not needed. Do we need a mutator method in this case?
Benefits of encapsulation

- Protects an object from unwanted access by clients.
  - Example: If we write a program to manage users' bank accounts, we don't want a malicious client program to be able to arbitrarily change a 
    BankAccount object's balance.
- Allows you to change the class implementation later.
- As a general rule, all instance data should be modified only by the object, i.e. instance variables should be declared private.

Access Protection: Summary

Access protection has three main benefits:
- It allows you to enforce constraints on an object's state.
- It provides a simpler client interface. Client programmers don't need to know everything that's in the class, only the public parts.
- It separates interface from implementation, allowing them to vary independently.

General guidelines

As a rule of thumb:
- Classes are public.
- Instance variables are private.
- Constructors are public.
- Getter and setter/mutator methods are public.
- Other methods must be decided on a case-by-case basis.

Printing Objects

- We would like to be able to print a Java object like this:
  ```java
  Student student = new Student(...);
  System.out.println("student: " + student);
  ```
- Would like this to provide output that is more useful than what Java provides by default.
  - Need to provide a toString() method

The toString() method

- tells Java how to convert an object into a String
- called when an object is printed or concatenated to a String:
  ```java
  Point p = new Point(7, 2);
  System.out.println("p: " + p);
  ```
- Same as:
  ```java
  System.out.println("p: " + p.toString());
  ```
- Every class has a toString(), even if it isn't in your code.
  - The default is the class's name and a hex (base-16) hash-code:
  ```java
  Point@9e8c34
  ```
  - Example: toString() method for our Student class:
  ```java
  public String toString() {
      return "name: " + name + "\n" + "id: " + id + "\n" + "average: " + getAverage();
  }
  ```

toString() implementation

- code that returns a suitable String;
- Example: toString() method for our Student class:
Variable shadowing

• A method parameter can have the same name as one of the instance variables:
  public class Point {
    private int x;
    private int y;
    // this is legal
    public void setLocation(int x, int y) {
      // when using x and y you get the parameters
    }
  }

  Instance variables x and y are shadowed by parameters with the same names.

Avoiding variable shadowing

public class Point {
  private int x;
  private int y;
  ...
  public void setLocation(int x_value, int y_value) {  
    x = x_value;
    y = y_value;
  }
}

Avoiding shadowing using this

public class Point {
  private int x;
  private int y;
  ...
  public void setLocation(int x, int y) {
    this.x = x;
    this.y = y;
  }
}

Inside the setLocation method,
• When this.x is seen, the instance variable x is used.
• When x is seen, the parameter x is used.

Constructors and this

• One constructor can call another using this:
  public class Point {
    private int x;
    private int y;
    public Point() {
      this(0, 0); // calls the (x, y) constructor
    }
    public Point(int x, int y) {
      this.x = x;
      this.y = y;
    }
    ...
  }

Multiple constructors

• It is legal to have more than one constructor in a class.
• The constructors must accept different parameters.
  public class Point {
    private int x;
    private int y;
    public Point() {
      x = 0;
      y = 0;
    }
    public Point(int x, int y) {
      this.x = x;
      this.y = y;
    }
  }

Summary of this

• this: A reference to the current instance of a given class
• using this:
  • To refer to an instance variable: this.variable
  • To call a method: this.method(parameters);
  • To call a constructor from another constructor: this(parameters);
Example of using this

```java
public class MyThisTest {
    private int a;
    public MyThisTest() {
        this(42);
    }
    public MyThisTest(int a) {
        this.a = a;
    }
    public void someSomething() {
        int a = 1;
        System.out.println(a);
        System.out.println(this.a);
        System.out.println(this);
    }
    public String toString() {
        return "MyThisTest a=" + a; // refers to the instance variable a
    }
}
```

The implicit parameter

- During the call `die.roll();`,
  the object referred to by `die` is the implicit parameter to the method.
- The method `int roll()` is really `int roll(Die this)`
- The call `die.roll()` is translated to `roll(die)`

Method overloading

- Can you write different methods that have the same name?
  - Yes!

```java
System.out.println("I can handle strings");
System.out.println(1 + 2);
System.out.println(3.14);
System.out.println(object);
Math.max(10, 15);   // returns integer
Math.max(10.5, 15.0); // returns double
```

Useful when you need to perform the same operation on different kinds of data.

The return value is not part of the signature

- You **cannot** overload on the basis of the return type (because it can be ignored)

```java
Example of invalid overloading:
public int convert(int value) {
    return 2 * value;
}
public double convert(int value) {
    return 2.54 * value;
}
```

Method overloading

```java
public int sum(int num1, int num2){
    return num1 + num2;
}
public int sum(int num1, int num2, int num3){
    return num1 + num2 + num3;
}
```

- A method’s name + number, type, and order of its parameters: **method signature**
- The compiler uses a method's signature to bind a method invocation to the appropriate definition

Example

```java
Consider the class Pet
class Pet {
    private String name;
    private int age;
    private double weight;
    ...
}
```
Example (cont)

```java
public Pet()
public Pet(String name, int age, double weight)
public Pet(int age)
public Pet(double weight)
```

Suppose you have a horse that weights 750 pounds then you use:
```java
Pet myHorse = new Pet(750.0);
```
but what happens if you do:
```java
Pet myHorse = new Pet(750);
```

### Primitive Equality
- Suppose we have two integers \(i\) and \(j\)
- How does the statement \(i == j\) behave?
  - \(i == j\) if \(i\) and \(j\) contain the same value

### Object Equality
- Suppose we have two pet instances \(pet1\) and \(pet2\)
- How does the statement \(pet1 == pet2\) behave?
  - \(pet1 == pet2\) is true if both refer to the same object
  - The \(==\) operator checks if the addresses of the two objects are equal
  - May not be what we want!

### Object Equality
- Consider the following lines of code:
  ```java
  String s1 = new String("Java");
  String s2 = new String("Java");
  ```
- Is \(s1 == s2\) True?
  - a) Yes  b) No

### Object Equality
- Consider the following lines of code:
  ```java
  String s1 = new String("Java");
  String s2 = new String("Java");
  ```
- Is \(s1.equals(s2)\) True?
  - a) Yes  b) No
Object Equality - extended

- If you want a different notion of equality define your own `.equals()` method.
- Do `pet1.equals(pet2)` instead of `pet1==pet2`
- The default definition of `.equals()` is the value of `==` but for Strings the contents are compared.

equals for the Pet class

```java
public boolean equals (Object other) {
    if (!other instanceof Pet) {
        return false;
    }
    Pet otherPet = (Pet) other;
    return ((this.age == otherPet.age) &&
            (Math.abs(this.weight - otherPet.weight) < 1e-8) &&
            this.name.equals(otherPet.name));
}
```

This is not explained correctly in the book (section 5.3)!!

---

Naming things

- Computer programs are written to be read by humans and only incidentally by computers.
- Use names that convey meaning
- Loop indices are often a single character (i, j, k), but others should be more informative.
- Importance of a name depends on its scope: Names with a “short life” need not be as informative as those with a “long life”
- Read code and see how others do it