O-O Design, Static vs Non-static fields, Final variables, Constructors

This lecture looks at the Die class from Section 4.1 of the Lewis textbook and improves it slightly.

If we want to model a 6-sided game die, we need to create a class (all Java coding is a class), and we need the properties of:

1) rolling a die

2) getting a die face / value

3) setting the die face

4\*) changing the number of sides of the die

Everything we model in Java is going to be a class.

All ways to interact with the class will need public methods.

For each behavior/property, we need to create an appropriate method. First, we can determine the necessary method headers:

public int getFaceValue() {

public int roll() {

First, for faceValue, we need a way to "remember" the current face value so that calling the method always returns the proper value.

- Remember that the only way to remember anything is through a variable.

- Where should the variable be? We want the value to be retained for everytime we call the method, so we need to declare the variable outside the method.

- In particular, this will be an "instance variable" or "instance field": each instance of the Die class will have a variable saving its current face value.

Object-oriented design: for best results (exactly why will be explained later), we should make the field private but make the method that accesses it public.

- So all code that will use the face value must use the public method.

private int faceValue;

public int getFaceValue() {

return this.faceValue;

}

(Please note that the "this." is not needed. If omitted, Java automatically adds it, but it is included here for completeness.)

Note that there is no confusion with having a method and a field of the same name because they are used differently.

Also, we should add comments above each method and field so that someone reading the code understands the purpose:

/\* returns the current face value \*/

public int getFaceValue() {

return this.faceValue;

}

/\* sets the value of the die \*/

public void setFaceValue(int faceValue) {

this.faceValue = faceValue;

}

Note that the parameter variable and the instance field have the same name "faceValue". Java allows local variables inside methods to have the same name as fields.

Inside the method, faceValue refers to the closest definition (the parameter). If we wanted the field, we needed to use this.faceValue.

Java does not allow two fields with the same name nor two local variables (including input parameter variables) with the same name.

(Also note, that setFaceValue allows any value to be set. A better solution would be to use an if statement so that if the value is between 1 and 6,

set the new face value, otherwise keep the original face value.)

For the roll, we use the random() method of the Math class:

We looked at the random() method in the Java API and see that it takes no input and returns a random double value in the range [0.0, 1.0). A little math and

knowing our types lets us convert the value in [0.0, 1.0) to a value in {1, 2, 3, 4, 5, 6}.

public void roll() {

faceValue = this.faceValue = (int)(Math.random() \* 6.0) + 1;

return faceValue;

}

As noted in lecture, we can combine the two steps because the value of the assignment expression is the value assigned:

public void roll() {

return faceValue = this.faceValue = (int)(Math.random() \* 6.0) + 1;

}

Initial field values

Note that we never initialized the field faceValue. This seems to violate the second law of variables (you must assign a value for the first use of a variable).

However, fields are given default initial values. The default value is one of (0, false, null).

Fields are the only variables to get default values. It is a good idea to not rely on the default value but to explicitly set it to a reasonable initial value.

In our case, the value of 0 for a face is not a reasonable value. A better solution is to use 1.

private int faceValue = 1;

Magic Numbers

The use of 6.0 is not good. It is a "magic number". A "magic number" is a number that has a special meaning to the programmer.

Here, you need to know that 6 is the size of the die.

Magic numbers should be avoided and replaced with varables.

(Any number that is not 0 or 1 is usually magic.)

First attempt:

private static int numberOfSides = 6;

public int roll() {

return this.faceValue = (int)(Math.random() \* numberOfSides) + 1;

}

This stores numberOfSides as a class field. Why a class field? Because all die have the same size (right now) and so why make every instance have its own numberOfSides's field?

Static vs Non-static fields

There are two types of fields in Java:

1) instance variables (non-static fields): each instance has its own copy of the field.

2) class variables (static fields): there is a single copy of the field owned by the class and all instances of the class share the field.

Final fields

A better solution is to make numberOfSides a constant. The number of sides will never change.

When a variable is marked as final, its value - once assigned - is never changed.

(The Java convention is to use all capital letters for static final variables so they stick out in the code, but we are going to make it non-static soon so I do not follow the convention.)

private static final int numberOfSides = 6;

public int roll() {

return this.faceValue = (int)(Math.random() \* numberOfSides) + 1;

}

Constructors

Recall how the new operator works: Die d = new Die();

1) Allocates space for the object.

2) Initializes the object based on the inputs to new -> it does this by calling an appropriate constructor method with the given input.

3) Returns the address (memory location) for the object.

A constructor is a special method that is called by the new operator to initialize a class.

A constructor has the form: public ClassName(inputs) {

which is very similar to a normal method except that there is no return type and the name is identical to the class name.

A constructor is not inherited by classes that extend this class. Each class must define its own constructors.

Here is a good constructor to specify the size of the die:

public final int numberOfSides;

public Die(int numSides) {

numberOfSides = numSides;

}

(Note that numberOfSides is now not static so that every die can have its own size. However it is still final because a single die cannot change its size.)

Now we can use the constuctor to create different die of different sizes:

Die d8 = new Die(8);

Die d20 = new Die(20);

However, this code will now give an error: Die d6 = new Die();

It worked before, what happened?

IMPORTANT: If you do not define a constructor, Java provides a default one that takes no input.

Once you create a constructor for your class, you lose the default constructor.

It would be nice to still have a default constructor, so we must write our own.

Method Overloading: You are allowed to have multiple methods of the same name (including constructors) as long as the methods differ in their parameter signature (the number and types of the inputs).

Here is a default constructor:

public Die() {

numberOfSides = 6;

}

Why must the parameter signatures be different? Suppose you have two constructors that both take int. If you call

new Die(10)

how does Java know which one you want? In this case, we have two constructors, one takes int and one takes nothing so it is easy to tell which one the user wants:

new Die(10) <- call the constructor that takes an int

new Die() <- call the constructor that takes no input