Anatomy of an Object

September 10, 2007

ComS 207: Programming I (in Java)
Iowa State University, FALL 2007
Instructor: Alexander Stoytchev

Quick review of last lecture

Methods in The Random Class

```
Random ()
   Constructor: creates a new pseudorandom number generator.

float nextFloat ()
   Returns a random number between 0.0 (inclusive) and 1.0 (exclusive).

int nextInt ()
   Returns a random number that ranges over all possible int values (positive
   and negative).

int nextInt (int num)
   Returns a random number in the range 0 to num-1.
```


Random Example

```
import java.util.Random;
...

Random generator = new Random();

int num = generator. nextInt();

float num2 = generator.nextFloat();
```


Math Class

```
static int abs (int num)
   Returns the absolute value of num.

static double acos (double num)

static double asin (double num)

static double atan (double num)
   Returns the arc cosine, arc sine, or arc tangent of num.

static double cos (double angle)

static double sin (double angle)

static double tan (double angle)
   Returns the angle cosine, sine, or tangent of angle, which is measured
   in radians.

static double ceil (double num)
   Returns the ceiling of num, which is the smallest whole number greater
   than or equal to num.

static double exp (double power)
   Returns the value e raised to the specified power.

static double floor (double num)
   Returns the floor of num, which is the largest whole number less than
   or equal to num.

static double pow (double num, double power)
   Returns the value num raised to the specified power.

static double random ()
   Returns a random number between 0.0 (inclusive) and 1.0 (exclusive).

static double sqrt (double num)
   Returns the square root of num, which must be positive.
```


Math Example

```
value = Math.abs(total) + Math.pow(count, 4);
```

Methods in NumberFormat Class

`String format (double number)`
Returns a string containing the specified number formatted according to this object's pattern.

`static NumberFormat getCurrencyInstance()`
Returns a NumberFormat object that represents a currency format for the current locale.

`static NumberFormat getPercentInstance()`
Returns a NumberFormat object that represents a percentage format for the current locale.

NumberFormat Example

```
double dollars=5.994;
NumberFormat fmt = NumberFormat.getCurrencyInstance();
System.out.println ( "Price = " + fmt.format(dollars) );

RESULT:
Price = $5.99
```


Methods in DecimalFormat Class

`DecimalFormat (String pattern)`
Constructor: creates a new DecimalFormat object with the specified pattern.

`void applyPattern (String pattern)`
Applies the specified pattern to this DecimalFormat object.

`String format (double number)`
Returns a string containing the specified number formatted according to the current pattern.

DecimalFormat Example

```
double miles =  .5395;

DecimalFormat fmt = new DecimalFormat("0.###");
System.out.println ( "Miles = " + fmt.format(miles) );

RESULT:
Miles = 0.540

Miles = 0.54
```


TestFormat.java example

Wrapper Classes

- The `java.lang` package contains *wrapper classes* that correspond to each primitive type:

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean
void	Void

Integer Class

```
Integer (int value)
   Constructor: creates a new Integer object storing the specified value.

byte byteValue ()
double doubleValue ()
float floatValue ()
int intValue ()
long longValue ()
   Return the value of this Integer as the corresponding primitive type.

static int parseInt (String str)
   Returns the int corresponding to the value stored in the
   specified string.

static String toBinaryString (int num)
static String tohexString (int num)
static String toOctalString (int num)
   Returns a string representation of the specified integer value in the
   corresponding base.
```


Autoboxing Examples

```
Integer obj1;
int num1 = 69;
obj1 = num1;    // automatically creates an
                //integer object


Integer obj2= new Integer(69);
int num2;
num2 = obj2;    // automatically extracts
                //the int value
```


TestInteger.java example

Enumerated Types (Section 3.7)

Enumerated Types

- Java allows you to define an enumerated type, which can then be used to declare variables
- An enumerated type establishes all possible values for a variable of that type
- The values are identifiers of your own choosing
- The following declaration creates an enumerated type called Season

```
enum Season {winter, spring, summer, fall};
```

- Any number of values can be listed

Enumerated Types

- Once a type is defined, a variable of that type can be declared

```
Season time;
```

and it can be assigned a value

```
time = Season.fall;
```

- The values are specified through the name of the type
- Enumerated types are *type-safe* – you cannot assign any value other than those listed

Ordinal Values

- Internally, each value of an enumerated type is stored as an integer, called its *ordinal value*
- The first value in an enumerated type has an ordinal value of zero, the second one, and so on
- However, you cannot assign a numeric value to an enumerated type, even if it corresponds to a valid ordinal value

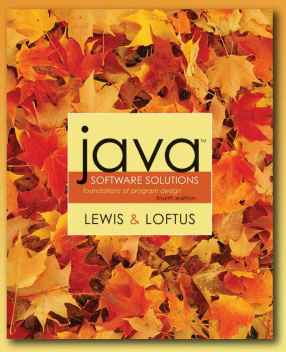
Enumerated Types

- The declaration of an enumerated type is a special type of class, and each variable of that type is an object
- The `ordinal` method returns the ordinal value of the object
- The `name` method returns the name of the identifier corresponding to the object's value
- See IceCream.java (page 137)

Run IceCream.java (page 137) in the textbook

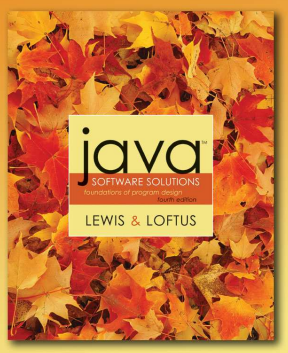
Chapter 4

Writing Classes



Chapter 4

Sections 4.1 & 4.2



Writing Classes

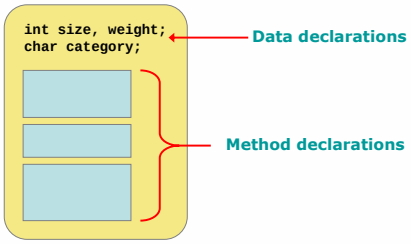
- The programs we've written in previous examples have used classes defined in the Java standard class library
- Now we will begin to design programs that rely on classes that we write ourselves
- The class that contains the `main` method is just the starting point of a program
- True object-oriented programming is based on defining classes that represent objects with well-defined characteristics and functionality

Classes and Objects

- Recall from our overview of objects in Chapter 1 that an object has *state* and *behavior*
- Consider a six-sided die (singular of dice)
  - It's state can be defined as which face is showing
  - It's primary behavior is that it can be rolled
- We can represent a die in software by designing a class called `Die` that models this state and behavior
  - The class serves as the blueprint for a die object
- We can then instantiate as many die objects as we need for any particular program

Classes

- A class can contain data declarations and method declarations



Classes

- The values of the data define the state of an object created from the class
- The functionality of the methods define the behaviors of the object
- For our `Die` class, we might declare an integer that represents the current value showing on the face
- One of the methods would "roll" the die by setting that value to a random number between one and six

Classes

- We'll want to design the `Die` class with other data and methods to make it a versatile and reusable resource
- Any given program will not necessarily use all aspects of a given class
- See RollingDice.java (page 157)
- See Die.java (page 158)

The Die Class

- The `Die` class contains two data values
  - a constant `MAX` that represents the maximum face value
  - an integer `faceValue` that represents the current face value
- The `roll` method uses the `random` method of the `Math` class to determine a new face value
- There are also methods to explicitly set and retrieve the current face value at any time

The toString Method

- All classes that represent objects should define a `toString` method
- The `toString` method returns a character string that represents the object in some way
- It is called automatically when an object is concatenated to a string or when it is passed to the `println` method

- System.out.println ("Die One: " + die1 + ", Die Two: " + die2);

Constructors

- As mentioned previously, a *constructor* is a special method that is used to set up an object when it is initially created
- A constructor has the same name as the class
- The `Die` constructor is used to set the initial face value of each new die object to one
- We examine constructors in more detail later in this chapter

Data Scope

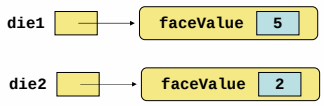
- The *scope* of data is the area in a program in which that data can be referenced (used)
- Data declared at the class level can be referenced by all methods in that class
- Data declared within a method can be used only in that method
- Data declared within a method is called *local data*
- In the `Die` class, the variable `result` is declared inside the `toString` method -- it is local to that method and cannot be referenced anywhere else

Instance Data

- The `faceValue` variable in the `Die` class is called *instance data* because each instance (object) that is created has its own version of it
- A class declares the type of the data, but it does not reserve any memory space for it
- Every time a `Die` object is created, a new `faceValue` variable is created as well
- The objects of a class share the method definitions, but each object has its own data space
- That's the only way two objects can have different states

Instance Data

- We can depict the two `Die` objects from the `RollingDice` program as follows:



Each object maintains its own `faceValue` variable, and thus its own state

Run examples from the book

THE END