CS1004: Intro to CS in Java, Spring 2005

Lecture #13: Java OO cont'd.

Janak J Parekh
janak@cs.columbia.edu

Administrivia

- Homework due next week
  - Problem #2 revisited

Constructors, revisited

- Remember: a constructor has *no return type* specified in the method header, not even `void`
  - A common error is to put a return type on a constructor, which makes it a "regular" method that happens to have the same name as the class
- The programmer does not have to define a constructor for a class
  - Each class has a *default constructor* that accepts no parameters

Defining the Die class

- Goal: design the `Die` class with other data and methods to make it a versatile and *reusable* resource
- That said, it doesn't mean a program *has* to use all the features of a class
- Let's write out what a possible Die class might be
  - An int that has the face value
  - Methods to roll and set the die explicitly
  - Methods to get info on the die's current value

DieRoller class

- Once we've defined a `Die`, we need to actually *use* it somehow
- We'll define a class called `DieRoller`, in which we'll actually manipulate the die
- This is a common model
  - Define one or more *data* classes
  - Establish one or more *program* classes, with a `main` method

Variables and "scope"

- As you may have guessed, there's multiple places to put variables in your program (scope)
  - At the class level (*instance* variables)
  - Inside a method (*local* variables)
- Variables declared inside one method *cannot* be used in another method without being explicitly *passed* to it
- What happens when you declare a variable with the same name in two places?

The "toString" Method

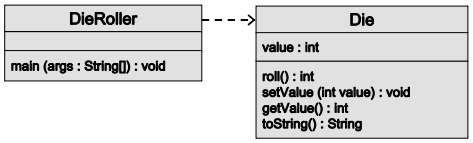
- All classes that represent objects should define a `toString` method
- The `toString` method returns a character string that represents the object in some way
- It is called automatically when an object is concatenated to a String or when it is passed to the `println` method

UML Diagrams

- UML stands for the *Unified Modeling Language*
- *UML diagrams* show relationships among classes and objects
- A UML *class diagram* consists of one or more classes, each with sections for the class name, attributes (data), and operations (methods)
- Lines between classes represent *associations*
- A dotted arrow shows that one class *uses* the other (calls its methods)

UML Class Diagrams

- A UML class diagram for our example
- We're not going to explore this too deeply, just enough for basic diagramming



Encapsulation

- We can take one of two views of an object:
  - *internal* - the details of the variables and methods of the class that defines it
  - *external* - the services that an object provides and how the object interacts with the rest of the system
  - "Box" metaphor
- From the external view, an object is an *encapsulated* entity, providing a set of specific services
- These services define the *interface* to the object

Object-oriented design

- One object (called the *client*) may use another object for the services it provides
- The client of an object may request its services (call its methods), but it should not have to be aware of how those services are accomplished
- Any changes to the object's state (its variables) should be made by that object's methods
- We should make it difficult, if not impossible, for a client to access an object's variables directly
- Not a strict requirement, but generally considered good design

Visibility Modifiers

- In Java, we accomplish encapsulation through the appropriate use of *visibility modifiers*
- A *modifier* specifies particular characteristics of a method or data (`final`)
- Java has three visibility modifiers: `public`, `protected`, and `private`
- The `protected` modifier involves inheritance, which we will discuss later

Visibility Modifiers, cont'd.

- *Public visibility:* can be referenced anywhere
- P*rivate visibility:* can be referenced only within that class
- No visibility modifier is *default visibility*, and can be referenced by any class in the same package
- An overview of all Java modifiers is presented in Appendix E
- So what's their preferred use?

Visibility for Variables

- Public variables violate encapsulation because they allow the client to "reach in" and modify the values directly
- Therefore instance variables should generally not be declared with public visibility
- It is acceptable to give a constant public visibility – although the client can access it, its value cannot be changed

Visibility for Methods

- Methods that provide the object's services (*service methods*) are declared with public visibility so that they can be invoked by clients
- Methods to assist service methods (*support methods*) are not intended to be called by a client and should not be declared with public visibility

Visibility Modifiers: Summary

	`public`	`private`
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

Accessors and Mutators

- If you want to let a client access data in a class, provide *accessor and mutator methods*
- The names of accessor and mutator methods usually take the form `getX` and `setX`, respectively, where `X` is the name of the value
- Sometimes called "getters" and "setters"
- The use of mutators gives the class designer the ability to restrict a client's options to modify an object's state
  - For example, restrict setting the value of a `Die` to a valid range

Enumerated Types

- If you're defining a class just to store one basic property, consider using an *enumerated type* instead
- An enumerated type establishes all possible values for a variable of that type; 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
- Specify type of Die:

```
enum DieType {weighted, fair};
```

- No instantiation needed:

```
DieType dt = DieType.weighted;
```

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
  - For *type safety* purposes
- 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

Let's do one more example

- Let's create two geometric shapes, circle and square, and play with them briefly

Next time

- Finish GUIs
- Start chapter 5 of L/L