4A. Writing Classes I

- Objectives - when we have completed this set of notes, you should be familiar with:
  - Anatomy of a class
  - Instance data
  - UML class diagrams
  - Encapsulation
  - Anatomy of a method
  - Parameters
  - Local data
  - Constructors

Writing Classes

- Thus far you have written programs that use classes defined in the Java standard class library
- The driver program (programs with a main method) should not contain all of your code
- Object-oriented programming:
  - Classes define sets of objects that will hold data and have specified behavior
  - Each class should be contained a separate file
  - Separate files facilitate testing

Classes and Objects

- An object has a state and a behaviors
  - You have used the Scanner class, which was written for the Java API

```
Scanner input = new Scanner(System.in);
```

  - It's **state** includes the "source" for the Scanner object (e.g., System.in); what input/data will be "scanned"
  - It's **behaviors** include reading the next line, reading the next integer, etc.

```
input.nextLine();
```

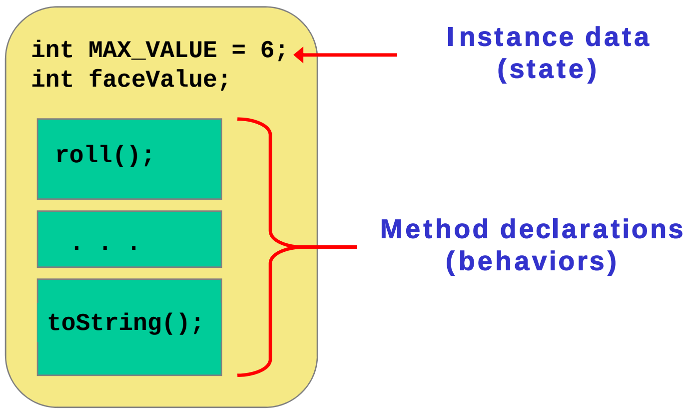

Classes and Objects

- Consider a six-sided die (singular of dice)
- Its state might include a face value (the value 1-6 that is currently showing)
- Its behaviors might include…
  - roll (roll the die to a random value 1-6)
  - setFaceValue (set the die to a specified value 1-6)
  - getFaceValue (get the face value)
- Example of how the Die class could be used:

```
Die dieObj = new Die();
dieObj.roll();
int rollResult = dieObj.getFaceValue();
```

Classes

- You would have to create a Die class with instance data for the state and methods for the behaviors



Classes

- You can now create multiple dice in one program
- A program will not necessarily use all aspects of a given class
- See RollingDice.java (page 162)
- See Die.java (page 165)

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
  - Called automatically anytime the object is referenced where a String is needed (e.g., when concatenated to a string or when it is passed to the `println` method)
  - In the jGRASP Interactions pane, `toString` is called automatically when an object reference is evaluated as an expression (e.g., if Die1 is a reference for a Die object, then entering Die1 in interactions evaluates to the result of its `toString` method)

Constructors

- Recall, 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 in the Die class

Data Scope

- The *scope* of data is the enclosing context in a program where that data can be referenced (used)
- *Instance data* (declared at the top of the class) can be referenced by all methods in that class - - scope is the entire class
- *Local data* (declared inside of a method) can only be used within that method
  - Example: 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

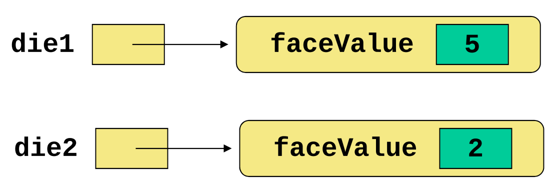
Instance Data

- 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
- Each time a `Die` object is created using the `new` operator, a new `faceValue` variable is created within the object
- All objects of a class will use the same code in their methods, but each object has its own data space for instance variables

COMP 1210 – Fundamentals of Computing I   Slide 4A - 11

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

Q

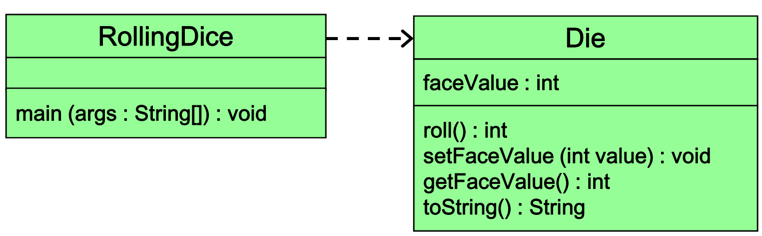
COMP 1210 – Fundamentals of Computing I   Slide 4A - 12

UML Diagrams

- UML stands for the *Unified Modeling Language*
- *UML class 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 (e.g., calls its methods)

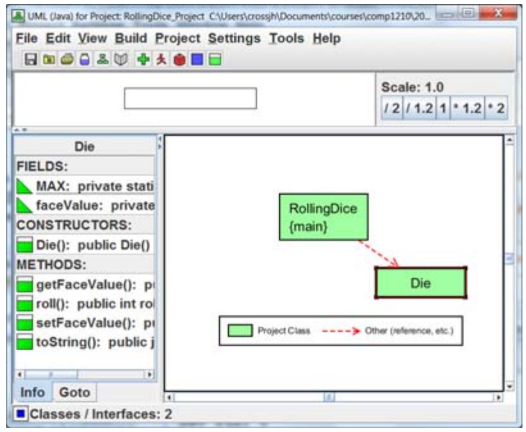
UML Class Diagrams

- A UML class diagram for the `RollingDice` program:



UML Class Diagrams in jGRASP

- Generate UML Class Diagram
- Select the Die class
- Right-click, select “Show Class Info”
- Info tab shows fields, constructors, and methods



Encapsulation

- One object (the client) can access and modify another object’s state through it’s methods. Example:

  `dieObj.setFaceValue(6);`

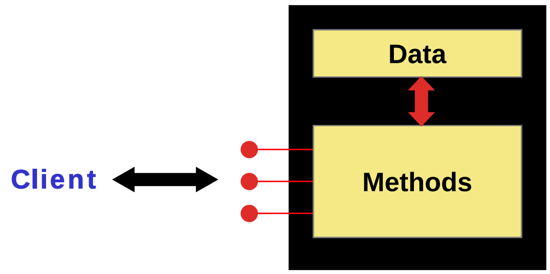
- We should make it difficult, if not impossible, for a client to access an object’s variables directly

  `dieObj.faceValue = 6;`

- The second line should cause a compile-time error; otherwise, the class violates *encapsulation*

Encapsulation

- An encapsulated object can be thought of as a *black box* -- its inner workings are hidden from the client
- The client invokes the interface methods of the object, which manages the instance data



Visibility (Access) Modifiers

- How do we make sure not to violate encapsulation?
- *visibility modifiers* define who can access an instance variable or a method
- Java has three visibility modifiers: `public`, `protected`, and `private`
- The `protected` modifier involves inheritance, which we will discuss later

Visibility (Access) Modifiers

- Members of a class that are declared with *public visibility* can be referenced anywhere
  - public instance variables violate encapsulation
- Members of a class that are declared with *private visibility* can be referenced only within that class
  - For now, all instance variables should be private
- Members declared with no visibility modifier have *default visibility* - - can be referenced by classes in the same package
- See Appendix E for more information

Visibility (Access) Modifiers

- Public methods are also called *service methods*, and offer useful behaviors to the client
- Sometimes methods get too large or multiple methods need to use the same code
  - *Support methods* are declared as private and can only be used by other methods in the class
  - Example: the Scanner class has many support methods, but many are not useful to the user or may cause unexpected behaviors if misused
- The Java API shows only public members (fields, constructors, and methods)

Visibility (Access) Modifiers

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

Q

Accessors and Mutators

- Because instance data is private, a class has methods to access and modify data values
- An *accessor method* returns the current value of a variable (sometimes called a "getter")
  - Example: getFaceValue in Die
- A *mutator method* changes the value of a variable (sometimes called a "setter")
- Example: setFaceValue in Die
- The names of accessor and mutator methods take the form getX and setX, respectively, where X is the name of the field

Mutator Restrictions

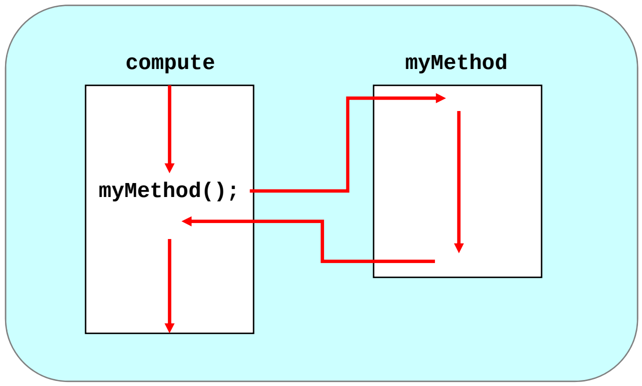
- Choose carefully which variables have getter and setter methods
  - If you had to sort through accessor methods for every instance variable in String or Scanner, it would make the class hard to use
  - Sometimes you don't want the user to be able to change a piece of data at all, and so no mutator method is created
- In Chapter 5, we'll see how to set other restrictions on fields (e.g., a field can only take on values in specified range)

Method Declarations

- A *method declaration* specifies the code that will be executed when the method is invoked (called)
- When a method is invoked, the flow of control jumps to the method and executes its code
- When complete, the flow returns to the place where the method was called and continues
- The invocation may or may not return a value, depending on how the method is defined

Method Control Flow

- If the called method is in the same class, only the method name is needed



Method Header

- A method declaration begins with a *method header*

```
char calc (int num1, int num2, String message)
```

method name

parameter list

return type

The parameter list specifies the type and name of each parameter

The name of a parameter in the method declaration is called a *formal parameter*

Method Body

- The method header is followed by the *method body*

```
char calc (int num1, int num2, String message)
{
   int sum = num1 + num2;
   char result = message.charAt (sum);

   return result;
}
```

The return expression must be consistent with the return type

sum and result are local data

They are created each time the method is called, and are destroyed when it finishes executing

The return Statement

- The *return type* of a method indicates the type of value that the method sends back to the calling location
- A method with a void return type returns no value
- A *return statement* specifies the value that will be returned

return *expression*;

Q

Parameters

- When a method is called, the *actual parameters* [arguments] in the invocation are copied into the *formal parameters* in the method header

```
ch = obj.calc (4, count, "War Eagle!");
```

```
char calc (int num1, int num2, String message)
{
   int sum = num1 + num2;
   char result = message.charAt (sum);

   return result;
}
```

MethodExample.java

Local Data

- As we've seen, local variables can be declared inside a method
- The formal parameters of a method create *automatic local variables* when the method is invoked
- When the method finishes, all local variables are destroyed (including the formal parameters)
- Keep in mind that instance variables, declared at the class level, exists as long as the object exists (i.e., a variable references the object)

End

Constructors Revisited

- Note that 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
- If the programmer does not define a constructor, then the class has a *default constructor* that accepts no parameters

Q