4. Writing Classes

- Objectives - when we have completed this set of notes, you should be familiar with:
    - Anatomy of a class: state and behaviors
    - Constructors
    - UML class diagrams
    - Encapsulation
    - Anatomy of a method: Parameters, Local data
    - Constant fields (public and private)
    - Invoking methods in the same class
    - Building a class incrementally
    - Testing a class
    - Writing a driver program

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 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 "target" 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();
    ```
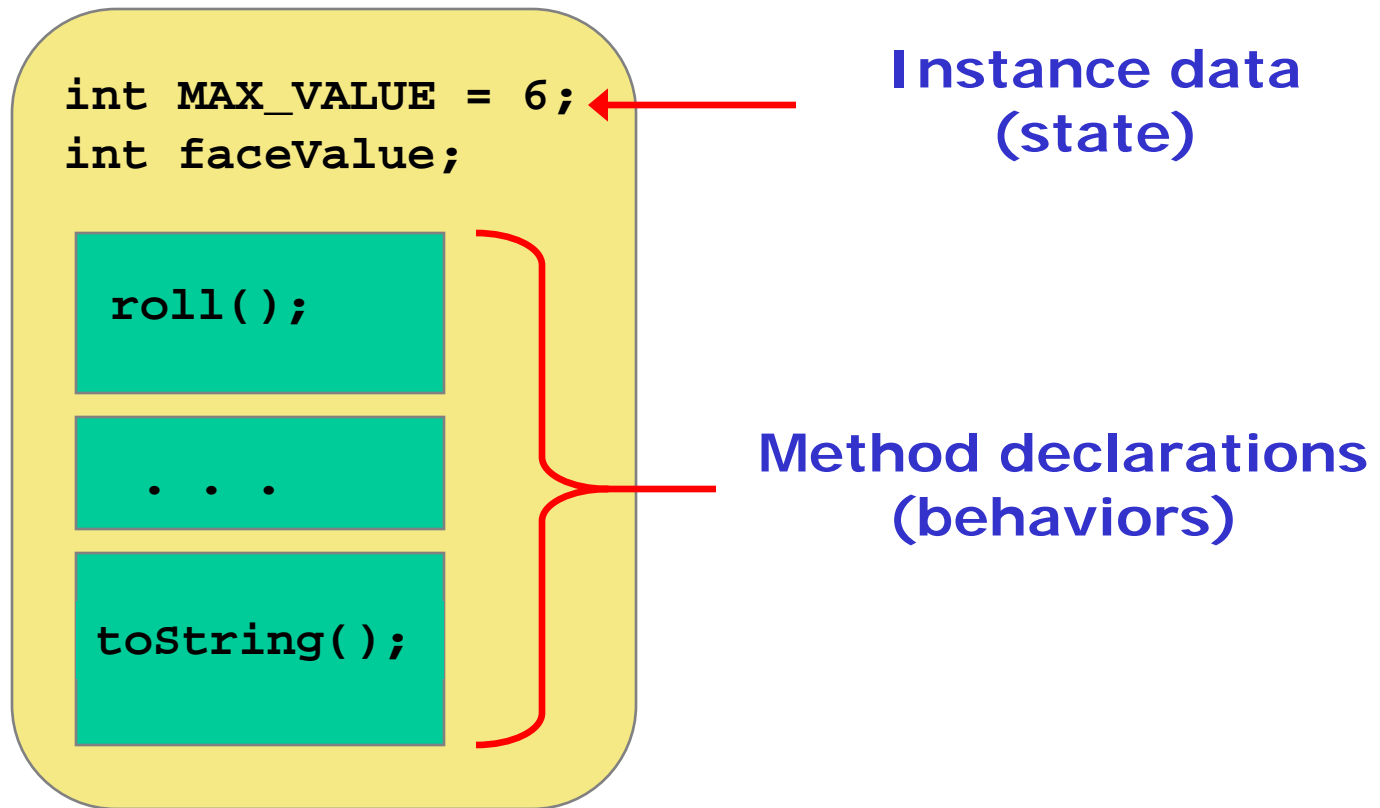
Classes and Objects

- 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

```
int MAX_VALUE = 6;
int faceValue;

roll();

. . .

toString();
```

Instance data (state)

Method declarations (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 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 create an object and set its initial state

- A constructor has the <u>same name as the class</u>, <u>no return type</u>, and may or may not have parameters

- The `Die` constructor has no parameters but sets the initial face value of each new die object to one

Data Scope

- The *scope* of data is the enclosing context in a program where that data can be referenced (used)

- *Instance data* (fields 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

- The declaration of an instance variable specifies 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

Instance Data

- We can depict the two `Die` objects from the `RollingDice` program as follows:
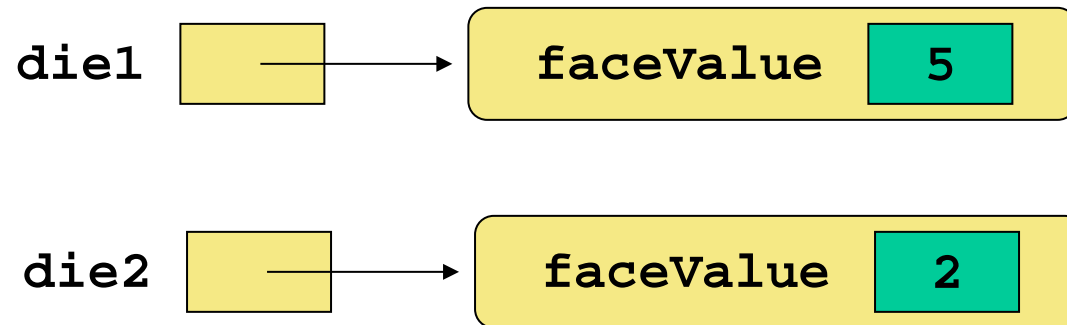
Each object maintains its own `faceValue` variable, and thus its own state

- jGRASP – canvas Object viewer:


Q

UML Class 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; a simple diagram has only the class name whereas a detailed diagram also has sections for the attributes (data) and operations (methods)

- Lines between classes represent *associations;* A dashed line arrow shows that one class *uses* the other (e.g., calls its methods)
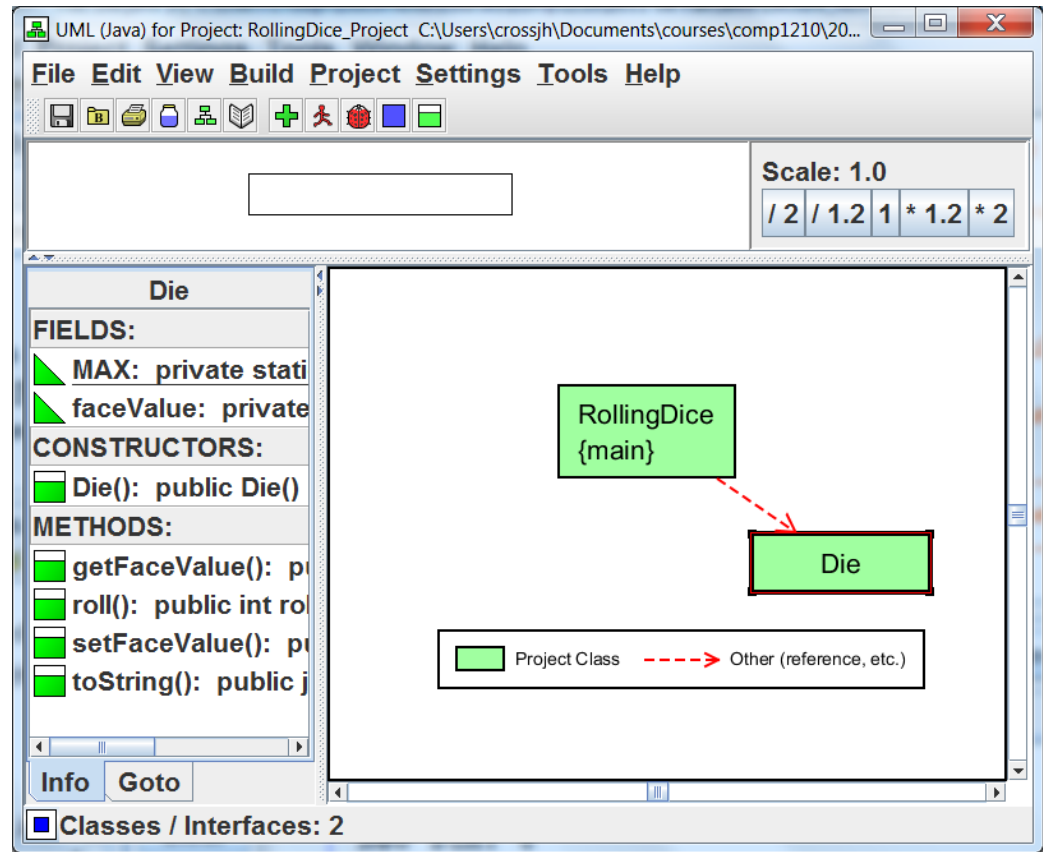
UML Class Diagrams

- A UML class diagram for the `RollingDice` program:
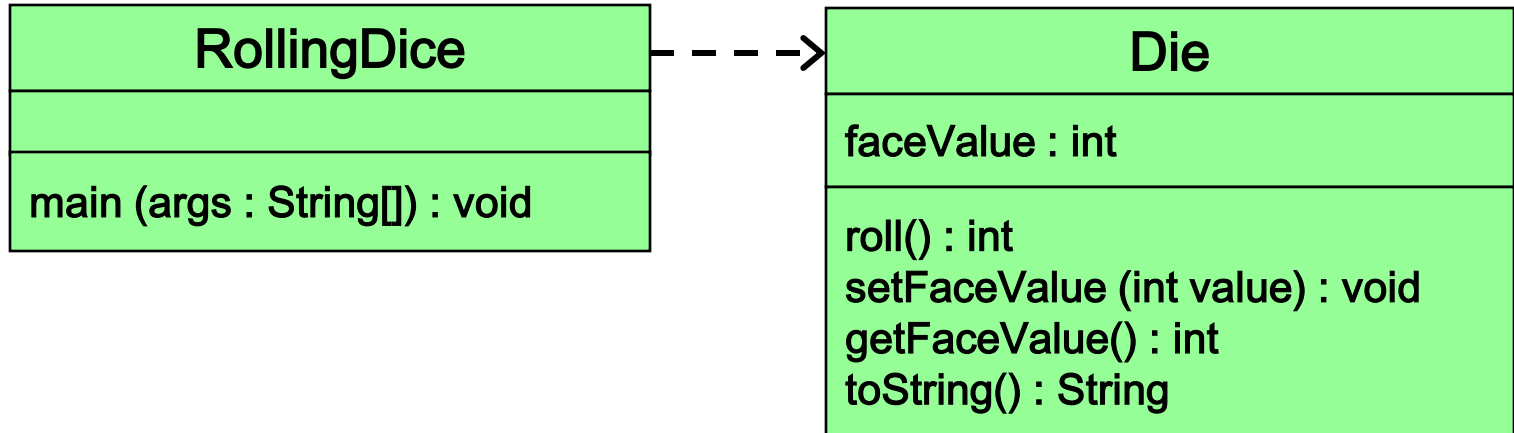
UML Class Diagrams in jGRASP

- 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
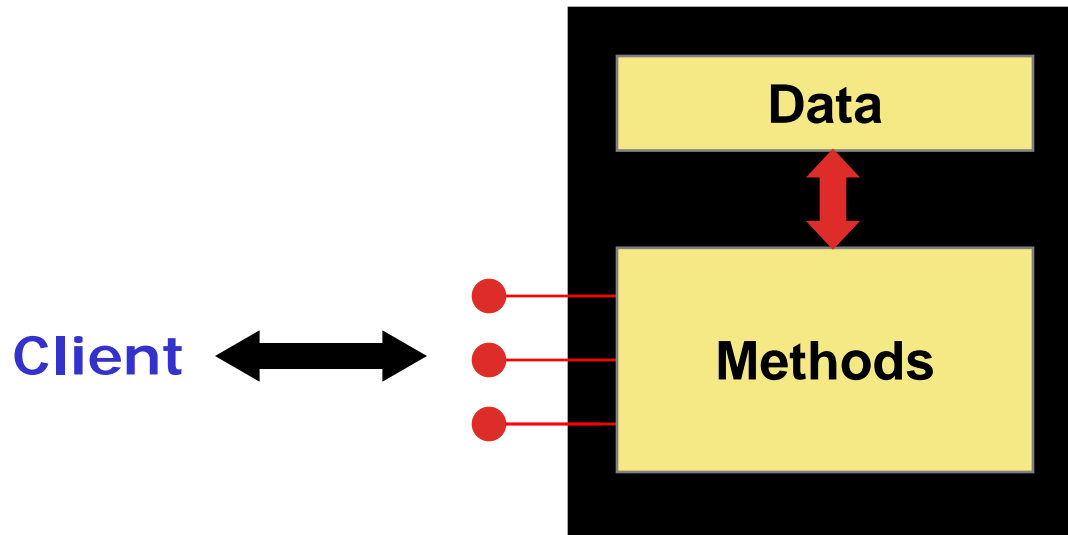
  ```
  dieObj.faceValue = 6;
  ```

- 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

Access (Visibility) Modifiers

- How do we make sure not to violate encapsulation?

- *Access modifiers* define who can access an instance variable or a method

- Java has three access modifiers:
  `public`, `protected`, and `private`

- The `protected` modifier involves inheritance, which we will discuss later

- The fields, constructors, and methods of a classes (collectively called *members* of the class) can each have an access modifier

Access (Visibility) Modifiers

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

Access (Visibility) Modifiers

- *Service methods* are public methods that offer useful behaviors to the client

- *Support methods* are declared as private and can only be used by other methods in the class

  - Sometimes methods get too large or multiple methods need to use the same code

  - Solution: create a private method for use by other methods in the class (e.g., calculations needed by several methods could placed in a separate method)

- The Java API shows only public members (fields, constructors, and methods)

Access (Visibility) 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

  - While many classes that you write will have at least some getter and setter methods, others may not have any

  - Sometimes you don't want the user to be able to get or set fields, so no accessor or mutator methods are provided (e.g., the String and Scanner classes have no getters or setters)

- Use if-else statements (Ch 5) to to set <u>mutator restrictions </u>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
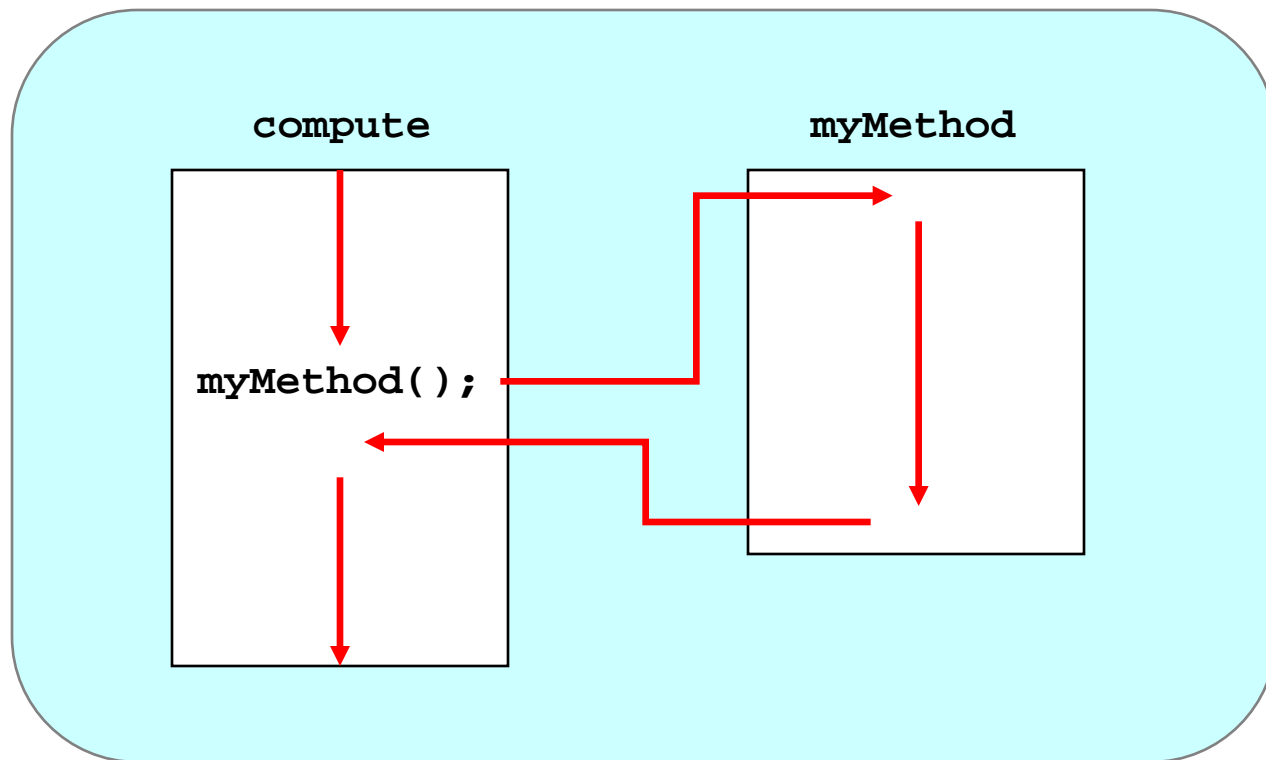
In the debugger, "Step-In" to the called method

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

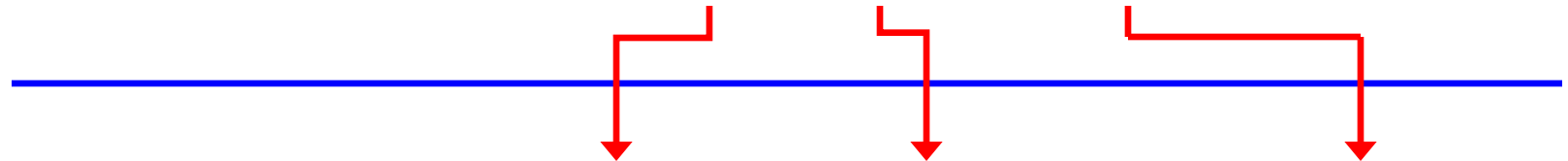- A **return** *statement* specifies the value that will be returned

```
return expression;
```

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)

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

Constant Fields

- Recall that a class constant can be declared as follows:

  ```
  private static final int MAX_AMOUNT = 5;
  ```

  - Uses the static and final reserved words (static will be covered in detail later in the course)

  - The name is in all caps with words separated by _

- In general, you do not want "magic numbers" (literal values) in your code.

  Replace this code: `if (number > 5) . . .`

  With this code: `if (number > MAX_AMOUNT)`
                                        . . .

Constant Fields

- Unlike variables, constants can be public without violating encapsulation.

  - For example, the PI constant in the Math class.

    ```
    double circumference = Math.PI * diameter;
    ```

  - Referenced by using the name of the class, the dot operator, and the name of the constant.

  - When you define a constant inside your own class, you can reference it inside the class without using the class name.

    ```
    if (number > MAX_AMOUNT). . .
    ```

Constant Fields

- Suppose the Student class had two types of students: undergrad and graduate.

```
public static final int GRADUATE = 0, UNDERGRAD = 1;
```

- Now client programs can set student type using constants (studentObj is an instance of Student):

Replace this code:

```
studentObj.setStudentType(0);
```

With this code:

```
studentObj.setStudentType(Student.GRADUATE);
```

Building a Class

- Suppose you wanted to create a class called Loan representing a loan with a balance, interest, and maximum loan amount
  - The balance starts at 0
  - The interest is 0.05 unless set otherwise
  - There are two loan types: employees and customers
  - Employees can borrow up to $100,000
  - Customers can borrow up to $10,000

- Create the empty class:

```
public class Loan {


}
```

Building a Class

- The constructor for the bank class should take **an int parameter** representing the type of loan (employee or customer)

- Create an empty constructor; you will fill in the code later:

```
public class Loan {

    public Loan(int accountType) {

    }

}
```

Building a Class

- Create an empty method for each of the following methods:

  - A 'getter' method for the balance (a double):

    ```
    public double getBalance() {

    }
    ```

  - A 'setter' method for the interest rate (a double) that returns true and sets the interest only if the interest is from 0 to 1:

    ```
    public boolean setInterest(double newInterest) {

    }
    ```

Building a Class

- Create an empty method for each of the following additional methods:
  - borrow: adds an additional amount to the loan and returns true only if the loan stays below the maximum amount.

```java
public boolean borrow(double amount) {

}
```

  - calculateTotalBalance: returns a double representing the loan amount with the interest added.

```java
public double calculateTotalBalance() {

}
```

Building a Class

- Now you have method stubs for each one of your methods, but your program will not compile.

  - If a method does not have a void return type, then you must return something or it will not compile.
  - For now, just include a placeholder return – any value of that type. Examples:

```java
public double getBalance() {
  return 0;
}

public boolean setInterest(double newInterest) {
  return false;
}
```

Building a Class

- You can now compile your program with empty methods. See the [Loan class in examples/method_stubs](#) for an example.

- This is the point where you would submit your project to the Skeleton Code assignment in Web-CAT.

Building a Class: Variables

- Add instance variables to the class. Take another look at the class description:
  - Suppose you wanted to create a class called Loan representing a loan with a **balance**, **interest**, and **maximum loan amount**
    - **The balance starts at 0**
    - The interest is 0.05 unless set otherwise
    - There are two loan types: employees and customers
    - Employees can borrow up to $100,000
    - Customers can borrow up to $10,000

```
private double balance = 0, interestRate,
        maxLoanAmount;
```

Building a Class: Constants

- Look for values that could be represented as constants:
  - **The interest is 0.05 unless set otherwise**
  - There are two loan types: employees and customers
  - **Employees can borrow up to $100,000**
  - **Customers can borrow up to $10,000**

The default interest rate and maximum loan amounts will be important for the class, but won't be needed by client programs. Therefore, the constants are private.

```java
private static final double DEFAULT_INTEREST = 0.05;
private static final double CUSTOMER_MAX = 10000;
private static final double EMPLOYEE_MAX = 100000;
```

Building a Class: Constants

- Look for values that could be represented as constants:
  - The interest is 0.05 unless set otherwise
  - **There are two loan types: employees and customers**
  - Employees can borrow up to $100,000
  - Customers can borrow up to $10,000

  The loan type (customer or employee) will be set in the constructor using an int value. It would therefore be useful to provide constants for the client program:

  ```java
  public static final int EMPLOYEE_ACCOUNT = 0;
  public static final int CUSTOMER_ACCOUNT = 1;
  ```

Building a Class: Constructor

- The constructor should set the interest rate to the default interest and then set the maximum loan amount based on the parameter.

- To test your constructor, instantiate objects in the interactions pane and check the values of the instance variables in the workbench (next slide).

Building a Class: Constructor

- The constructor should set the interest rate to the default interest and then set the maximum loan amount based on the parameter.
    - To test your constructor, instantiate objects in the interactions pane and check the values of the instance variables in the workbench.

```
Loan customer = new Loan(Loan.CUSTOMER_ACCOUNT);

Loan empl = new Loan(Loan.EMPLOYEE_ACCOUNT);
```

Building a Class: Methods

- You can now complete your getBalance method and setInterest method.
    - In general, **get methods will not change the state of the object**, but will only return a value.
    - Set methods will often have a boolean return type to indicate if the inputs are valid and the method actually "set" the field.
    - Test each one of your methods as you create them. You can view each method's effect on instance variables using interactions and the workbench.

Building a Class: Testing

- Example: Test both the function and the return of the setInterest method (workbench and interactions).

```
Loan loanObj = new Loan(Loan.CUSTOMER_ACCOUNT);
loanObj.setInterest(-1);
loanObj.setInterest(-1)
false
loanObj.setInterest(2)
false
loanObj.setInterest(0.5)
true
```

See the completed Loan class in Loan.java

Repeated Code

- In general, you do not want to repeat calculations in your code.
  - The toString method should return a string that includes the total balance.
  - In this case, make a call to the method that performs the calculation:

```java
public String toString() {
    String output = "Loan amount: "
        + balance + "\r\n"
        + "Total balance with interest: $"
        + calculateTotalBalance()
        + "\r\n";
    return output;
}
```

Driver Program

- The Loan class cannot be run as an application, but can only be used by other programs.

- In this case, we might want to create a driver program that users can execute to set up a loan, borrow an amount, and view their balance.

- See LoanCalculator.java for an example of a driver program for the Loan class.

Summary

- Objectives - you should now be familiar with:
  - Anatomy of a class: state and behaviors
  - Constructors
  - UML class diagrams
  - Encapsulation
  - Anatomy of a method: Parameters, Local data
  - Constant fields (public and private)
  - Invoking methods in the same class
  - Building a class incrementally
  - Testing a class
  - Writing a driver program