CSI1102
Introduction to Software Design

Chapter 4:
Writing Classes

Learning objectives: Writing Your Own Classes

- Understand what the following entails:
  - An object, its *state* and *behavior*
  - Class definitions
  - Encapsulation and Java modifiers
  - About methods:
    - method declaration, invocation, and parameter passing
    - method overloading
    - method decomposition
  - Object relationships and aggregation
  - graphics-based objects

What is an Object?

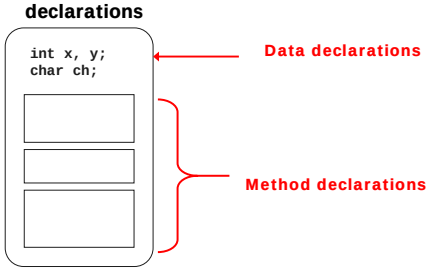
- An object has:
  - *state* - descriptive characteristics
  - *behaviors* - what it can do (or what can be done to it)
- For example, consider a **coin that can be flipped** so that it's face shows either "heads" or "tails"
  - The state of the coin is its current face (heads or tails)
  - The behavior of the coin is that it can be flipped
  - Note that the behavior of the coin might change its state

What are Classes?

- A *class* is a blueprint of an object
- It is the model or pattern from which objects are created
- For example, the `String` class is used to define `String` objects
  - Each `String` object contains specific characters (its state)
  - Each `String` object can perform services (behaviors) such as `toUpperCase`
- We can also write our own classes that **define specific objects that we need**
- We can write a `Coin` class to represent a coin object

More about Classes

- A class contains **data declarations** and **method declarations**



An example: The Coin Class

- Data:
  - `face`, an integer that represents the current face
  - `HEADS` and `TAILS`, integer constants that represent the two possible states
- Methods:
  - a `Coin` **constructor**, to **initialize the object**
  - a `flip` method, to flip the coin
  - a `isHeads` method, to determine if the current face is heads
  - a `toString` method, to return a string description for printing

The Coin Example: CountFlips.java

```
// Demonstrate the use of a programmer-defined class

public class CountFlips
{
    //
    public static void main (String[] args)
    {
        final int NUM_FLIPS = 1000;
        int heads = 0, tails = 0;

        Coin myCoin = new Coin(); // instantiates the Coin object
```

Continued...

The Coin Example: CountFlips.java

```
    for (int count = 1; count <= NUM_FLIPS; count++)
    {
        myCoin.flip( );

        if (myCoin.isHeads())
            heads++;
        else
            tails++;
    }

    System.out.println("Number of flips" + NUM_FLIPS);
    System.out.println("Number of heads" + heads);
    System.out.println("Number of tails" + tails);
}
}
```


The Coin Example: Coin.java

```
public class Coin
{
    public final int HEADS = 0;
    public final int TAILS = 1;

    private int face;

    // ----------------------------------------------
    //   Sets up the coin by flipping it initially.
    //
     public Coin ()
    {
       flip();
    }
```

Continued...

The Coin Example: Coin.java

```
    // ----------------------------------------------
    //   Flips the coin by randomly choosing a face.
    //
     public void flip()
    {
       face = (int) (Math.random() * 2);
    }

    // ----------------------------------------------
    //   Returns true if the current face is heads.
    //
    public boolean isHeads()
    {
            return (face == HEADS);
    }
```

public static double random() // p.870 Math class in java.lang
// returns a random number between 0.0 and 1.0

Continued...

The Coin Example: Coin.java

```
 // ----------------------------------------------------
//   Returns the current face of the coin as a string
//
    public String toString()
        {
          String faceName;

          if (face == HEADS)
              faceName = "Heads";
          else
              faceName = "Tails";

          return faceName;
        }
    }
```


An example: Running CountFlips

- Run 1:
  - Number of flips 1000
    Number of heads 493
    Number of tails 507
- Run 2:
  - Number of flips 1000
    Number of heads 507
    Number of tails 493

"Overriding" Static Methods: An example

- I added the following method to the **Coin class** and it worked:

```
public static double random()
{
    System.out.println("My random");
    return 0;
}
```

- You are not allowed to create a method in the Math class named random()
- Actually, you may be sued by SUN if you try to **extend / modify** Packages, APIs, Classes or Methods created by them
  - **they have the ownership!**

About the strengths/weaknesses of Static Classes and Methods

- Recall: You do not have to create an object using new
- They stay in the system even if you may not need them all the time
- They are not efficiently managed by the automatic garbage collection
  - Too many static classes and methods may slow down your application
  - They are an "inheritance" from C++ and actually do not really fit in "object oriented" paradigm

A Discussion: The Coin Class

- Note that the `CountFlips` program did not use the `toString` method
- A program **will not necessarily use every service provided by an object**
- Once the `Coin` class has been defined, we can use it again in other programs as needed

Data Scope and Instance Data

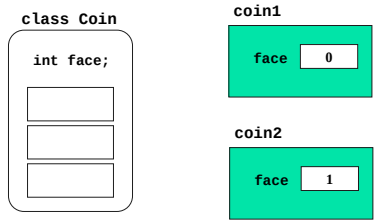
- The *scope* of data is the area in a program in which that data can be used (referenced)
- Data declared at the class level can be used by all methods in that class
- *Local Data* declared within a method can be used only in that method

Instance Data

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

Instance Data

See FlipRace.java (page 217)



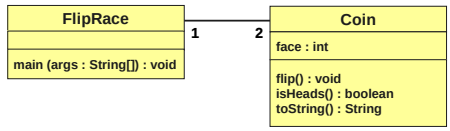
UML Diagrams

- 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, and methods
- Lines between classes represent *associations*
- *Associations* can show *multiplicity*

19

UML Class Diagrams

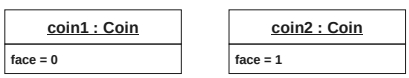
- A UML class diagram for the `FlipRace` program:



20

UML Diagrams

- A UML *object diagram* consists of one or more instantiated objects.
- It is a snapshot of the objects during an executing program, showing data values



21

Object property: Encapsulation

- We can take one of two views of an object:
  - internal - the variables the object holds and the methods that make the object useful
  - external - the services that an object provides and how the object interacts
- From the external view, an object is an *encapsulated* entity, providing a set of specific services
- These services define the *interface* to the object
- Recall from Chapter 2 that an object is an *abstraction*, hiding details from the rest of the system

22

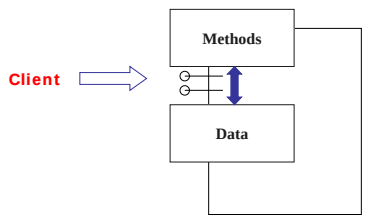
More about Encapsulation

- An object should be *self-governing*
- Any changes to the object's state (its variables) should be made only by that object's methods
- We should make it difficult, if not impossible, to access an object's variables other than via its methods
- The user, or *client*, of an object can request its services, but it should not have to be aware of how those services are accomplished

23

More about Encapsulation

- An encapsulated object can be thought of as a *black box :* Its inner workings are hidden to the client, which invokes only the interface methods



24

Visibility Modifiers

- In Java, we accomplish encapsulation through the appropriate use of *visibility modifiers*
- A *modifier* is a Java reserved word that specifies particular characteristics of a method or data value
- We've used the modifier `final` to define a constant
- Java has three visibility modifiers: `public`, `protected`, and `private`
- The `protected` modifier involves inheritance, which we will discuss later

Visibility Modifiers

- Members of a class that are declared with *public visibility* can be accessed from anywhere
- Public variables violate encapsulation
- Members of a class that are declared with *private visibility* can only be accessed from inside the class
- Members declared without a visibility modifier have *default visibility* and can be accessed by any class in the same package
- Java modifiers are discussed in detail in Appendix F

More about Visibility Modifiers

- Methods that provide the object's services are usually declared with public visibility so that they can be invoked by clients
- Public methods are also called *service methods*
- A method created simply to assist a service method is called a *support method*
- Since a support method is not intended to be called by a client, it should not be declared with public visibility

Visibility Modifiers

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

Driver Programs

- A *driver program* drives the use of other, more interesting parts of a program
- Driver programs are often used to test other parts of the software
- The `Banking` class contains a `main` method that drives the use of the `Account` class, exercising its services
- See Banking.java (page 226)
- See Account.java (page 227)

Driver Programs: Banking.java

```
public class Banking
{
// creates some bank accounts and requests some services
        public static void main (String[] args)
        {
          Account acct1 = new Account("Joe Smithfield", 2341, 200.00);
          Account acct2 = new Account("Sue Smith", 3212, 1300.01);

          acct1.deposit(23.43);

          double smithBalance = acct2.deposit(500.00);
          System.out.println("Smith's balance " + smithBalance);

          acct1.addInterest();

          System.out.println("Smithfields's new balance " + acct1.getBalance());
        }
}
```

Driver Programs: Account.java

```
public class Account
    {
        private final double RATE = 0.035; // interest rate of 3.5%

        private long acctNumber;
        private double balance;
        private String name;

        // Constructor sets up the account
        public Account (String owner, long account, double initial)
        {
                name = owner;
                acctNumber = account;
                balance = initial;
        }

Continued...
```


Driver programs: Account.java (cont).

```
public double deposit (double amount)
{
        if (amount < 0) // deposit value is negative
                System.out.println("Error: invalid deposit.");
        else
                balance = balance + amount;

        return balance;
}

public double addInterest()
{
        balance += (balance * RATE);
        return balance;
}

public double getBalance()
{
        return balance;
} }
```


The results: Banking & Accounts

- Smith's balance 1800.01
- Smithfield's new balance 231.25

Notes:

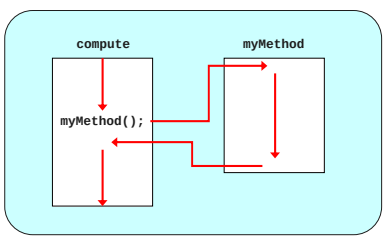
- E.g. The balance field in PRIVATE, we cannot “directly” access the data value; use a method.
- See p.227-229 for the entire program
- It all uses the NumberFormat class to format the output

Method Declarations

- A *method declaration* specifies the code that will be executed when the method is invoked (or 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

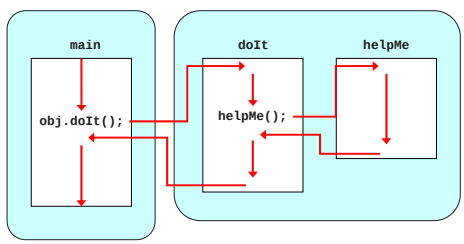
Method Control Flow

- The called method can be within the same class, in which case only the method name is needed



Method Control Flow

- The called method can be part of another class or object



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 argument*

Method Body

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

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

```
{
   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 that does not return a value has a `void` return type
- A *return statement* specifies the value that will be returned

```
return expression;
```

- Its expression must conform to the return type

Method Parameters

- Each time a method is called, the *actual parameters* in the invocation are copied into the formal parameters

```
ch = obj.calc (2, count, "Hello");
```

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

   return result;
}
```


Method Local Data

- 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
- Any method in the class can refer to instance data

Constructors Revisited

- **Recall that a constructor is a special method that is used to initialize a newly created object**
- When writing a constructor, remember that:
  - it has the same name as the class
  - it does not return a value
  - it has no return type, not even `void`
  - it typically sets the initial values of instance variables
- The programmer does not have to define a constructor for a class

Overloading Methods

- *Method overloading* is the process of using the same method name for multiple methods
- The *signature* of each overloaded method must be unique
- The signature includes the number, type, and order of the parameters
- The compiler determines which version of the method is being invoked by analyzing the parameters
- The return type of the method is not part of the signature

Overloading Methods

Version 1

```
float tryMe (int x)
{
   return x + .375;
}
```

Version 2

```
float tryMe (int x, float y)
{
   return x*y;
}
```

Invocation

```
result = tryMe (25, 4.32)
```


Overloaded Methods

- The `println` method is overloaded:

```
println (String s)
println (int i)
println (double d)
```

and so on...

- The following lines invoke different versions of the `println` method:

```
System.out.println ("The total is:");
System.out.println (total);
```


Overwriting Constructors: SnakeEyes.java

```
public class SnakeEyes
{
   // Roll dice and count snake eyes
   public static void main (String[] args)
   {
      final int ROLLS = 500;
      int snakeEyes = 0, num1, num2;

      Die die1 = new Die();      //creates a 6-sides die
      Die die2 = new Die(20);    //creates a 20-sides die

      for (int rols = 1; rols <= ROLLS; rols++)
      {
            num1 = die1.roll();
            num2 = die2.roll();

            if (num1 == 1 && num2 ==1)
                  snakeEyes++;
      }

      System.out.println("Number of Snake Eyes " + snakeEyes);
   }}
```


Overwriting Constructors: Die.java

```
public class Die
{ private final int MIN_FACES = 4;
  private int numFaces;   // number of sides on the die
  private int faceValue;  // current value showing on the die
  //-----------------------------------------------------------------
  //  Defaults to a six-sided die. Initial face value is 1.
  //-----------------------------------------------------------------
  public Die ()
   { numFaces = 6;     faceValue = 1; }
  //-----------------------------------------------------------------
  //  Explicitly sets the size of the die. Defaults to a size of
  //  six if the parameter is invalid.  Initial face value is 1
```


Overwriting Constructors: Die.java (continued)

```
  public Die (int faces)            //second constructor
  {  if (faces < MIN_FACES)
           numFaces = 6;
     else      numFaces = faces;
     faceValue = 1;
  }
        //  Rolls the die and returns the result.
  public int roll ()
  {  faceValue = (int) (Math.random() * numFaces) + 1;
     return faceValue;
  }
        //    Returns the current die value.
  public int getFaceValue ()
  { return faceValue;  }
}
```

Tips about writing methods: Method Decomposition

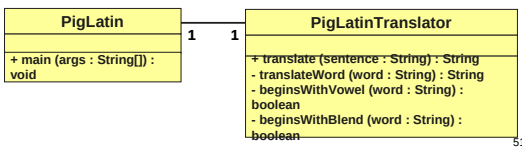
- A method should be relatively small, so that it can be understood as a single entity
- A potentially large method should be decomposed into several smaller methods as needed for clarity
- A service method of an object may call one or more support methods to accomplish its goal
- Support methods could call other support methods if appropriate

Pig Latin Example: "happy" becomes "appyhay"

- The process of translating an English sentence into Pig Latin can be decomposed into the process of translating each word
- The process of translating a word can be decomposed into the process of translating words that
  - begin with vowels
  - begin with consonant blends (sh, cr, tw, etc.)
  - begins with single consonants
- See PigLatin.java (page 238) and PigLatinTranslator.java (page 240)

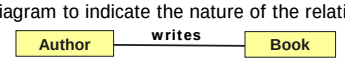
Class Diagrams Revisited

- In a UML class diagram, public members can be preceded by a plus sign
- Private members are preceded by a minus sign
- A class diagram for the `PigTranslator` program:



Object Relationships

- Objects can have various types of relationships to each other
- A general *association*, as we've seen in UML diagrams, is sometimes referred to as a *use relationship*
- A general association indicates that one object (or class) uses or refers to another object (or class) in some way
- We could even annotate an association line in a UML diagram to indicate the nature of the relationship



Object Relationships: Rational Number example

- Association may occur between objects of the same class
- For example, consider some `Rational` number objects:

```
Rational r1 = new Rational(6, 8);
Rational r2 = new Rational(1, 3);
Rational r3, r4, r5;

r3 = r1.add(r2);
r4 = r1. subtract(r2);
R5 = r1.divide(r2);
```

- One object (`r1`) is executing the method and another (`r2`) **is passed as a parameter**

Object relationships: How does "add" work?

```
public int getDenominator()
{
    return denominator;
}

public Rational add (Rational op2)
{
    // the calculation
    int commonDenominator = demonimator * ops.getDenominator();
    int numerator1 = numerator + ops.getDenominator();
    int numerator2 = op2.getNumerator() * denominator;
    int sum = numerator1 + numerator 2;

    return new Rational (sum, commonDenominator);
}
```

Aggregation

- An *aggregate object* is an object that contains references to other objects
- For example, an `Account` object contains a reference to a `String` object (the owner's name)
- An aggregate object represents a *has-a* relationship
- A bank account *has a* name
- Likewise, a student may have one or more addresses

Aggregation: Student addresses

```
public class StudentBody
{
    public static void main(String[] args)
    {
        Address school = new Address("Smithstreet 1", "Ottawa", "ON", 12312);
        Address jHome = new Address("Mytown 23", Quebec", "QC", 23156);
        Student john  = new Student("John", "Doe", jHome, school);

        System.out.println(john); // use toString to automatically print
    }
}
```

- See page 250 to 253

The output

```
John Doe Hometown 23 Quebec QC 231 Smithstreet 1 Ottawa ON 12312
```


Aggregation: Student.java

```
public class Student
{ private String firstName, lastName;
  private Address homeAddress, schoolAddress;
  // set up the student object
  public Student(String first, String last, Address home, Address school)
  {
        firstName = first;
        lastName = last;
        homeAddress = home;
        schoolAddress = school;
  }
  public String toString()          // return student object as a string
  {
    String result;
    result = firstName + " " + lastName + " " + homeAddress + " " +schoolAddress;

    return result;
  } }
```

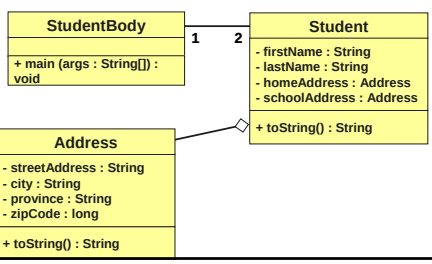

Aggregation: Address.java

```
public class Address
{ private String streetAddress, city, province;
  private long zipCode;
  // set up the address object
  public Address(String street, String town, String st, long zip)
  {
        streetAddress = street;
        city = town;
        province = st;
        zipCode = zip;
  }
  public String toString()          // return student object as a string
  {
    String result;
    result = streetAddress + " " + city + " " + province + " " + zipCode;

    return result;
  } }
```


Aggregation in UML

- An aggregation association is shown in a UML class diagram using an open diamond at the aggregate end



Applet Methods

- In previous examples we've used the `paint` method of the `Applet` class to draw on an applet
- The `Applet` class has several methods that are invoked automatically at certain points in an applet's life
- The `init` method, for instance, is executed only once when the applet is initially loaded
- The `start` and `stop` methods are called when the applet becomes active or inactive
- The `Applet` class also contains other methods that generally assist in applet processing

Graphical Objects

- Any object we define by writing a class can have graphical elements
- The object must simply obtain a graphics context (a `Graphics` object) in which to draw
- An applet can pass its graphics context to another object just as it can any other parameter
- See LineUp.java (page 257) and StickFigure.java (page 259) how to draw "Stick Figure"

Summary: Writing Your Own Classes

- Understand what the following entails:
  - An object, its *state* and *behavior*
  - Class definitions
  - Encapsulation and Java modifiers
  - About methods
    - method declaration, invocation, and parameter passing
    - method overloading
    - method decomposition
  - Object relationships and aggregation
  - Graphics-based objects