We've been using predefined classes. Now we will learn to write our own classes to define new objects.

This week we focus on:
- Objects: attributes, state and behaviour
- Anatomy of a Class: attributes and methods
- Classes as Types
- Creating new objects
- Parameter passing

An object has:
- state - descriptive characteristics
- methods - what it can do (or what can be done to it) » services, actions, behavior, ....

For example, consider a bank client with a checking and a savings account.

The state of the client is the balance of the checking and saving accounts.

Methods are withdrawal, deposit and transfer, querying the balance etc.

Some methods might change the state
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:

\[
\text{Class} \quad \rightarrow \quad \text{Object Variable} \quad \rightarrow \quad \text{Object}
\]

\[
\text{String } x = \text{"Bob"}; \quad \rightarrow \quad \text{State}
\]

• Each String object contains specific characters (its state)
• Each String object has methods such as toUpperCase:

\[
x = x.\text{toUpperCase}();
\]
• In the case of String, the methods don’t change the object itself; but this is very specific to strings

The String class was provided for us by the Java standard class library
• But we can also write our own classes that define specific objects that we need
• For example, suppose we wanted to write a bank program that manages the clients and their saving and checking accounts.
• We could write a Client class to represent client objects with the two associated accounts.

Part 2

The Anatomy of a Class

• A class can be considered to be a cardboard box containing items (called members in Java):
  – Constants
  – Variables
  – Methods
    • constructor methods (that help creating an object of the class)
    • other useful methods (withdraw, transfer)
    • possibly a main method
• Each item (data and method) in the box can be accessed and modified by using the DOT operator
Classes

- A class contains data declarations and method declarations (collectively called members of the class)

```
int x, y;
char ch;
```

Data declarations

Method declarations

```
Our cardboard box ~ the class
```

The Idea Behind A Class

- A class builds objects
- Each class, generally, represents a real thing, for example:
  - Class Client represents the properties and behaviour of a Client of a bank.
  - Object X of class Client represents an actual particular client.

Classes

- A client has a checking account and a savings account
  - Each is represented by its balance
- We can perform withdrawals, deposits, transfers…

```
double balCheckings;
double balSavings;
```

withdraw
deposit
transfer

Java Methods

- **Method**: A set of statements that build a logical unit of action.
  - Class method: (more about this later)
  - Instance method: (let us focus on this one today)
    - Any method that is invoked with respect to an instance of a class. Also called simply a method.
  - Many methods need input (e.g. System.out.println("xxx");)
    - The inputs of a method are called its parameters.
    - Each parameter is of a certain type
  - Many methods return output (e.g. scan.nextInt());
    - The output of a method is called its return value.
    - The return value is of a certain type
  - A method in Java does not have to return a value,
    - declare the return type as void (as in the main method).

```
String replace(char oldChar, char newChar)
```
Writing Methods

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

```java
public static void main(String args[])
{
    int x = 5;
    System.out.println(x);
}
```

<table>
<thead>
<tr>
<th>Method Header</th>
<th>Code Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>{</td>
<td>int x = 5; System.out.println(x); }</td>
</tr>
</tbody>
</table>

Method Invocation

- 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
- If the methods has a return value
  - we can assign this value to a variable of the appropriate type
  - we can use the method call as an operand in an expression

Method Calls

Syntax:

- OBJECT.METHOD(PARAMETERS);
- X = OBJECT.METHOD(PARAMETERS);

Method Locations

- Methods only exist within classes
- When you invoke a method, we say that the method is being called.
- Assume you are in main method of class X, then
  - you can call a method from another class Y
    - static method on class name (e.g. Math.abs(int i))
    - other methods on objects of class Y (e.g., scan.nextInt())
  - you can call other methods of class X
    - we haven’t seen this so far (comes later)
    - has slightly different syntax
Constructors

- When we create an object from a class the first thing we need to do is initialize all the member variables (the variables defined within the object).
- The constructor is the method used to do this.
- Constructors are optional. If not present then the member variables need to get initialized somewhere else.
- You can identify the constructor because it has no return type (not even `void`) and it has the same name as the class.
- Its parameters and code body behave in the same way as regular methods.
- Constructors are only invoked when you initially create the object.

```java
public class Client {
    private double balChecking;  // member variables
    private double balSavings;

    public Client (double checkingBalance, double savingsBalance) {
        balChecking = checkingBalance;
        balSavings = savingsBalance;
    }

    public boolean withdrawalChecking (double amount) {
        if (amount < 0 || balChecking < amount) {
            System.out.println("Incorrect amount");
            return false;
        } else {
            balChecking -= amount;
            return true;
        }
    }

    public double depositChecking(double amount) {
        balChecking += amount;
        return balChecking;
    }

    public double depositSavings(double amount) { // similar to depositChecking
        balSavings += amount;
    }

    public void transfer (char fromAccount, double amount) {
        switch(fromAccount) {
            case 'c':
                balChecking -= amount;
                balSavings += amount;
                break;
            case 's':
                balSavings -= amount;
                balChecking += amount;
                break;
            default:
                System.out.println("Incorrect input to transfer");
        }
    }

    public double balanceChecking () {
        return balChecking;
    }

    public double balanceSavings () {
        return balSavings;
    }
}
```

```java
public class Bank {
    public static void main (String[] args) {
        Client c1 = new Client(100,0);
        Client c2 = new Client(0,0);
        double amount;

        amount = c1.depositChecking(100);
        System.out.println("c1's checking is now: ", amount);
        c1.transfer('c',50);
        if (c2.withdrawalSavings(20)) {
            System.out.println("Withdrawal successful");
        } else {
            System.out.println("Withdrawal not successful");
        }

        System.out.println ("checking 1: " + c1.balanceChecking());
        System.out.println ("checking 1: " + c1.balanceSavings());
        System.out.println ("checking 2: " + c2.balanceChecking());
        System.out.println ("checking 2: " + c2.balanceSavings());
    }
}
```
The Client Class

- Once the Client class has been defined, we can use it again in other programs as needed.
- For instance, we have used it in the Bank program.
- However, the Bank program has not used all methods provided by the Client class.
- A program will not necessarily use every service provided by an object.

Part 2

Some Object Details

Instance Variables

- The balChecking and balSavings variables in the Client class are called instance variables because each instance (object) of the Client class has its own values for these variables.
- A class declares the type of the data, but it does not reserve any memory space for it.
- Every time a Client object is created, a new balChecking variable and a new balSavings variable is created as well.
- The objects of a class share the method definitions, but they have unique data space for their instance variables.
  - This allows two objects to have separate states.

Instance Data

- The variables balChecking and balSavings are declared as double in the Client class.
- c1 and c2 are two different instances of the Client class.
- Each instance has its own values for the variables.
- The objects are related through the class definition, but they have separate states.
Method Declarations Revisited

• A method declaration begins with a *method header*

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

• The parameter list specifies the type and name of each parameter
  – names can be freely chosen (similar to variable names)
• The names of parameters in the header are called *formal parameters*
• Formal parameters can be used in the method body in the same way variables are used

```
int sum = num1 + num2;
char result = message.charAt (sum);
return result;
```

The return expression must be consistent with the return type

Local Data

• A method can declare its own variables
• These variables are local to the method
• Local variables are created (memory allocated) each time the method is called and discarded when the method finishes execution
• This is different to member variables
  – Member variables are declared in the class but not inside any particular method
  – Member variables exist throughout the lifetime of an object

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
• The *return statement* specifies the value that will be returned
• Its expression must conform to the return type
Example

```java
public class Calc {
    ...,
    int add(int x, int y) {
        int sum = x + y;
        return sum;
    }
}
```

Describe the flow and result

```java
public static void main(String args[]) {
    int result;
    Calc mycalc = new Calc();
    result = mycalc.add(5, 2);
}
```

What would happen if the type was not int?

Parameters

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

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

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

Constructors Revisited

- Recall that a constructor is a special method that is used to set up 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 often sets the initial values of instance variables
- The programmer does not have to define a constructor for a class

Examples for Client

```java
public Client(double startChecking, double startSavings) {
    balChecking = startChecking;
    balSavings = startSavings;
}
```

```java
public Client() {
    checking = 0;
    saving = 0;
}
```
Private and Public

- In our example, we declared
  - member variables as **private**
  - Member methods as **public**
- In general, each member (variable, method) can be either declared **private** or **public**
  - **public**
    - the member can be accessed externally (from outside the object) using the DOT operator
  - **private**
    - the member cannot be accessed externally. Only during execution within the object can the member be accessed.

Accessing an instance Variable

- Assume **Client** declares its instance variables public
  ```java
  public double balChecking; //member variables
  public double balSavings;
  ```
- Assume the **Bank** has created a client
  ```java
  Client cl = new Client(0,0);
  ```
- There are two options to access the instance variables of cl:
  ```java
  double balance = cl.balChecking;
  vs.
  double balance = cl.balanceChecking();
  ```
  - In the first case, the `balChecking` variable is directly accessed via the DOT operator
  - In the second case, a **getter or accessor** method of the Client is called that returns the value of the variable

Modifying an Instance Variable

- There are two options to modify the data of the cl:
  ```java
  cl.balChecking = 100;
  vs.
  cl.depositChecking(100);
  ```
  - In the first case, the `balChecking` variable is directly modified. It is accessed via the DOT operator and a value is assigned to it.
  - In the second case, a **setter or mutator** method of the Client object is called that performs the modification

Encapsulation

- Most instance data should only be accessed via getter and setter methods
  - Guarantees data is only accessed through one way: easy to control
- In order to protect against direct access,
  - instance variables should be declared **private**
  - all access and modifications to variables should be done via getter and setter methods
- Constants might or might not be made public depending on the application
  - For instance, assume that each deposit and withdrawal is associated with a fee
    - we want to make sure that each modification of the balance includes the fees
Considering Fees

```java
public class Client {
   private double balChecking;  //member variables
   private double balSavings;
   public final double FEE = 1.5;
   
   public boolean withdrawalChecking (double amount) {
      if (amount < 0 || checking < amount) {
         System.out.println("Incorrect amount");
         return false;
      } else {
         balChecking -= amount + FEE;
         return true;
      }
   }
   
   public double depositChecking (double amount) {
      checking += amount - FEE;
   }
}
```

Private vs. Public Methods

- We declare methods that should be publicly accessible as `public`
  - they are the services
  - they are the *interface* with which objects of the class can be accessed and manipulated
- We might have some helper methods used for internal decomposition
  - they support other methods in the class
  - they should be declared `private`

Classes with and without Main

- So far, we have seen two types of classes
  - classes that contain
    - a main method, no instance data, no other methods
    - examples: bank, calculator, and nearly all classes we programmed so far
  - classes that contain
    - no main method, a set of other methods, maybe some instance data
    - examples: Client, Scanner and other library classes

Classes with `main`

- These are classes that typically start an application
- `main` is declared static and returns void
  - Also has a special input argument
  - The keyword static indicates that the method is a class method
  - It can be called without the requirement to instantiate an object of the class.
  - (Other methods can be static, too. For example the methods in the `Math` class)
- When we start a program (run in DrJava), the interpreter invokes the main method of the class.
- A class X that does not contain a `main` method cannot execute on its own. We need at least one class with a main in our application
Application

- In theory, each application could be written as one big Java class.
- However, it is better to split an application into different classes that handle different tasks or sub-concepts of the application.
- In this case a “starter” class with a `main` method starts the application, creates objects of other classes, and coordinates the execution of the application.

Pretty Printing

- A class often contains a method that provides a string representation of its variables.
- In Class Client
  ```java
  public String toString()
  {
    String check = "Balance Checking: " + balChecking + "\n";
    String save = "Balance Saving: " + balSavings + "\n";
    return(check+save);
  }
  
  In Class Bank
  System.out.println(c1.toString());
  ```

A funny example

- A cat class
  - a cat can be fed
  - feeding leads to mood swings
- A starter class
  - creates cats
  - feeds cats and observes behaviour

Cat.java

```java
public class Cat {
   private float weight;
   private int age;
   private boolean isFriendly;

   public Cat() {
      weight = 3.8f;
      age = 2;
      moodSwing();
   }

   public String toString(){
      String sWeight = "I weigh " + weight + " kg.\n";
      String sAge = "I'm " + age + " years old.\n";
      String sFriendly = (isFriendly)? "I'm the nicest cat in the world"
      : "One more step and I'll attack."
      return (sWeight+sAge+sFriendly);
   }

   public float feed(float food){
      weight += food;
      System.out.println("it wasn't Fancy Feast's seafood fillet...");
      wail();
      return weight;
   }

   private void wail() {
      System.out.println("Miiiiaaawwwwwww!");
      moodSwing();
   }

   private void moodSwing(){isFriendly = ((int)(Math.random()*2) == 0);  }
}
```

How does this work?

- A cat class
  - a cat can be fed
  - feeding leads to mood swings
- A starter class
  - creates cats
  - feeds cats and observes behaviour
**FeedTheCats.java**

```java
public class FeedTheCats {
    public static void main(String args[]) {
        Cat Frisky = new Cat();
        Cat Tiger = new Cat();

        System.out.println("Frisky: " + Frisky.toString());
        System.out.println("Tiger: " + Tiger.toString());
        System.out.println("We are about to feed the cats...");
        float newWeight = Frisky.feed(1.2f);
        System.out.println("Frisky should weigh " + newWeight + " kg.");
        newWeight = Tiger.feed(2.4f);
        System.out.println("Tiger should weigh " + newWeight + " kg.");
        System.out.println("Frisky: " + Frisky.toString());
        System.out.println("Tiger: " + Tiger.toString());
    }
}
```

**Method invocation within object**

- **Note:**
  - If a class or an object calls a method on another object referenced by a variable name, the call is:
    - `Variablename.methodname`
  - If an object calls a method on itself, only the method name needs to be written:
    - `wail();`

**Two ways to implement Calculator**

1. **Application style**
   - **Calculator class**
     - with methods for addition/division
     - no main method
   - **Starter class**
     - with main
     - creates a calculator object and uses it (the for loop in original calculator)

2. **Calculator class with object**
   - methods for addition/division
   - main method
     - Creates an object of itself
     - Has loop to ask input and redirect to other methods
Using Objects

- Sometimes an object has to interact with other objects of the same type
- For example, we might add two Rational number objects together as follows:

\[ r3 = r1.add(r2); \]

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

RationalNumbers.java

```java
public class RationalNumbers {
    public static void main (String[] args) {
        Rational r1 = new Rational(6, 8);  // What are we doing here?
        Rational r2 = new Rational(1, 3);

        System.out.println("First rational number: "+ r1);
        System.out.println("Second rational number: "+ r2);

        if (r1.equals(r2)) System.out.println("r1 and r2 are equal.");
        else System.out.println("r1 and r2 are NOT equal.");

        Rational r3 = r1.add(r2);
        Rational r4 = r1.subtract(r2);
        Rational r5 = r1.multiply(r2);
        Rational r6 = r1.divide(r2);

        System.out.println("r1 + r2: " + r3);
        System.out.println("r1 - r2: " + r4);
        System.out.println("r1 * r2: " + r5);
        System.out.println("r1 / r2: " + r6);
    }
}
```

Questions

- RationalNumbers.java used a class called Rational:
  - What do you think the member variables should be in order to represent rational numbers?
  - How would you write the constructor?
  - Assuming that the denominator is the same, how would you write the ADD method?
  - If the denominator was not the same, how would you write the ADD method?
  - Assuming the denominator is the same, how would you write the equal method?
Part 3
Thinking Like A Programmer

Why Objects?

• Manageability
  – Self-contained (all in a single class)
  – Shareable (import .class)
  – Security features (private, protected, public)

• Lifelike:
  – Maps to real-life entities

Manageability

• Programs tend to get very long, hard to debug and difficult to solve in one sitting
• The way to control this is to write small programs
• Large programs can be reduced to many little methods that are easy to debug… this is called method decomposition.

Method Decomposition

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

Let’s see an example…
**PigLatin.java**

```java
public class PigLatin {
    public static void main (String[] args) {
        String sentence, result, another;
        Scanner scan = new Scanner(System.in);
        do {
            System.out.println ("Enter a sentence (no punctuation):" );
            sentence = scan.nextLine();
            result = PigLatinTranslator.translate (sentence);
            System.out.println ("That sentence in Pig Latin is:");
            System.out.println (result);
            System.out.print ("Translate another sentence (y/n)? ");
            another = scan.nextLine();
        } while (another.equalsIgnoreCase("y"));
    }
}
```

What does this do?

**PigLatinTranslator.java**

```java
public class PigLatinTranslator {
    // Translates a sentence of words into Pig Latin.
    public static String translate (String sentence) {
        String result = "";
        sentence = sentence.toLowerCase();
        Scanner scan = new Scanner (sentence);
        while (scan.hasNext()) {
            result += translateWord (scan.next());
            result += " ";
        }
        return result;
    }
}
```

A potentially large program

Still decomposing…

- Notice that we have only completed a small part of the job
- We still need to program: `translateWord`

**translateWord**

```java
private static String translateWord (String word) {
    String result = "";
    if (beginsWithVowel(word))
        result = word + "yay";
    else if (beginsWithPrefix(word))
        result = word.substring(2) + word.substring(0,2) + "ay";
    else
        result = word.substring(1) + word.charAt(0) + "ay";
    return result;
}
```

Notice we are still putting off work until later ... decomposition

Using built-in methods to help us
And Finally

```java
private static boolean beginsWithVowel (String word) {
    String vowels = "aeiou";
    char letter = word.charAt(0);
    return (vowels.indexOf(letter) != -1);
}

private static boolean beginsWithPrefix (String str) {
    return (str.startsWith("bl") || str.startsWith("pl") ||
            str.startsWith("br") || str.startsWith("pr") ||
            str.startsWith("ch") || str.startsWith("sh") ||
            str.startsWith("cl") || str.startsWith("sl") ||
            str.startsWith("cr") || str.startsWith("sr") ||
            str.startsWith("dr") || str.startsWith("sr") ||
            str.startsWith("fl") || str.startsWith("st") ||
            str.startsWith("fr") || str.startsWith("th") ||
            str.startsWith("gl") || str.startsWith("tr") ||
            str.startsWith("gr") || str.startsWith("wh") ||
            str.startsWith("kl") || str.startsWith("wr") ||
            str.startsWith("ph") );
}
```

Describe what is going on here

When thinking about your problem…

- First: Think of the problem as a whole or think of it as you would solve it by hand without a computer
- Then: Try to divide the work you did into steps or parts
  – Each of these steps or parts could be a potential little program contained in a method
- Last: Think of the parameters and return values for these steps or parts

If more time, give problems to solve during class time