Classes and Methods
CS101, Mock

We have already been using classes in Java – as we have seen, a class corresponds to an object. Classes encourages good programming style by allowing the user to encapsulate both data and actions into a single object, making the class the ideal structure for representing complex data types.

For example, consider the following HelloWorld class:

```
class HelloWorld {
      public void printHello() {
            System.out.println("hello, world");
      }
}
```

This **defines the model for an object**. However, at this point we haven't created an actual **instance** of the object. As an analogy, consider the blueprint to construct a car. The blueprint contains all the information we need as to how the parts of the car are assembled and how they interact. This blueprint for the car corresponds to the class definition.

An instance of an object corresponds to the actual object that is created, not the blueprint. Using our car analogy, we might use the blueprint to create multiple cars. Each car corresponds to an instance, and might be a little different from one another (for example, different colors or accessories, but all using the same blueprint). In Java, an instance is created when we use the keyword "new":

```
HelloWorld  x;                // Defines variable x to be of type "HelloWorld"
x = new HelloWorld();         // Creates an instance of HelloWorld object
```

At this point, x refers to an instance of the class. We allocate space to store any data associated with the HelloWorld object.

Format to Define a Class

A class is defined using the following template. Essentially we can store two types of data in a class, variables (data members), and functions (methods). A simple format for defining a class is given below; we will add some enhancements to it shortly!

```
class className {
        // Define data members; i.e. variables associated with this class
        type varname1;
        type varname2;
        …
        type varnameN;

        // Define methods; i.e. functions associated with this class
        return_type  methodName1(parameter_list);
        return_type  methodName2(parameter_list);
        …
        return_type  methodNameN(parameter_list);
}
```

Data members are variables defined in the class.

Methods are functions defined in the class. A method is just a collection of code. You should already be familiar with one method, the **main** function. The parameter list is used to pass data to the method. Since a method is a function, we have to declare what type of value the function will return (e.g., int, float, long, etc.) If we don't want the method to return any value, we have a special type called *void*.

Class Variables – Data Members

We already know what variables are. Variables defined inside a class are called *member variables*, because they are members of a class. For this reason, some programmers like to use the convention of naming data member variables prefixed with m_ . For example, instead of:

```
int value;
```

Some programmers will instead use:

```
int m_value;
```

To differentiate that this variable is a special variable, i.e. it is a member variable.

Member variables are accessible from code defined inside the class. By default, member variables are also accessible from code defined outside the class. A bit later we'll see a way to control whether or not these variables should be accessible from outside the class.

Let's look at a simple class that contains only data members and no methods.

```
class Money {
        int m_dollars;
        int m_cents;
}
```

Now consider another class that creates objects of type Money:

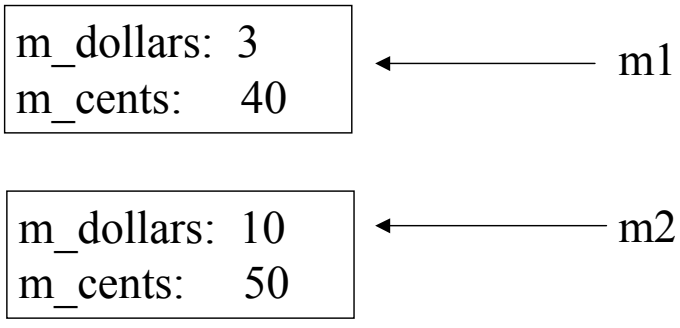
```
class Test {
        public static void main(String[] argv) {
                Money m1 = new Money();
                Money m2 = new Money();

                m1.m_dollars = 3;
                m1.m_cents = 40;
                m2.m_dollars = 10;
                m2.m_cents = 50;
                System.out.println(m1.m_dollars + " " + m1.m_cents);
                System.out.println(m2.m_dollars + " " + m2.m_cents);
        }
}
```

The output of this program is:

```
3   40
10  50
```

When the program reaches the print statement, we have created two separate instances of the Money object, each with different values stored in their member variables:



This can be quite convenient, because we can now associate multiple variables together in a single object. While both of these variables were of type integer in this example, the types could be anything. For example, a class to represent an Employee might contain variables like the following:

```
class Employee {
        String m_Name;
        int m_Age;
        double m_HourlyWage;
        long m_IDNumber;
}
```

In this example, we are associated different variable types with the Employee object. This is a powerful construct to help organize our data efficiently and logically.

Controlling Access to Data Members or Methods

As we saw with the money example, by default we have access from outside the class to any variables we define. We can explicitly state whether or not access is granted by using the keywords **public** or **private** (there is another keyword, **protected**, which we won't cover at this point).

To use these modifiers, prefix the class variable with the desired keyword. Public means that this variable can be accessed from outside the class. Private means that this variable is only accessible from code defined inside the class. This designation can be useful to hide data that the implementer of the class doesn't want the outside world to see or access.

For example if we redefine our Money class:

```
class Money {
        public int m_dollars;
        private int m_cents;
}

class Test {
        public static void main(String[] argv) {
                Money m1 = new Money();

                m1.m_dollars = 3;        // VALID, m_dollars is public
                m1.m_cents = 40;         // INVALID, m_cents is private
        }
}
```

This program will generate a compiler error since we are trying to access a private variable from outside the class. It is considered good programming style to always use public, private, or protected to indicate the access control of all class variables.

We can also apply the public and private modifiers to methods as well as variables, as we will see next. Note that these modifiers only apply to class variables, not to function variables (e.g., we won't use private/public on variables defined inside main).

Class Methods or Functions

So far, we have been working with relatively small programs. As such, the entire program has been coded up into a single method, **main**. For larger programs, a single method is often inconvenient and also hard to work with. The technique of dividing a program up into manageable pieces is typically done by constructing a number of smaller functions and then piecing them together as modules. This type of modularization has a number of benefits:

- Avoids repeat code (reuse a function many times in one program)
- Promotes software reuse (reuse a function in another program)
- Promotes good design practices (Specify function interfaces)
- Promotes debugging (can test an individual module to make sure it works properly)

Let's examine how to write programs using methods, also known as functions and sometimes referred to as procedures.

Before starting

We've already been using quite a few methods in our programs. Calls like println(), readString(), and readInt() are all methods that have been written by someone else that we're using. Note how the innards of these functions are all hidden from you – as long as you know the interface, or the input/output behavior, you are able to use these functions in your own programs. This type of data-hiding is one of the goals of functions and classes, so that higher-level code doesn't need to know the details of particular tasks.

Before starting and jumping into writing programs using functions, it is a good idea to look at the problem you are trying to address and see how it might logically be broken up. In this case, a block-diagram of the major code components can be a useful tool. As an example, consider a program to implement a grade book for a class. Some useful functions might be to:

- Input grades for each student
- Calculate the average grade
- Calculate the standard deviation
- Change a grade
- Store the grades to disk
- Load the grades from disk
- Print all grades

Some not-so-useful functions might be

- Add 1 to a student's grade
- Subtract 1 from a student's grade

The frequency of using these last two methods is so low that it is probably not worth the effort to create them. While this is somewhat of a trivial example, the point is that some thought and care should go into designing the functions before they are actually written. This step may actually take more time in creating a program than writing the actual code.

Defining a Method

To define a method use the following template:

```
modifier  return_type  functionName(type1  varName1,  type2  varName2, … )
{
   Instructions
   return (return_value);
}
```

Modifier is either public or private to indicate if this method is available from outside the class.

Return_type is a data type returned by the function. For example, int, float, long, another Class, or void if we have no data to return. If we are returning a value, we must specify what value the method returns with the return statement. Usually this is at the end of the function, but it could be anywhere inside the function.

functionName is an identifier selected by the user as the name for the function.

The list of parameters are input variables that are passed to the function from the caller. This gives data for the function to operate on. To define these parameters, specify the type of the variable followed by the variable name. Multiple parameters are separated by commas.

Here is a method that finds the maximum of three numbers and returns the max back, and some code in main that invokes this function:

```
class Foo {
        public int maximum(int num1, int num2, int num3)
        {
                int curmax;                     // Local variable, only exists
                                                // within this function!
                curmax = num1;
                if (num2 > num1) curmax = num2;
                if (num3 > curmax) curmax = num3;
                return (curmax);
        }

        public static void main(String[] args) {
                Foo x = new Foo();
                int max;

                max = x.maximum(5, 312, 55);            // Max gets 312
                System.out.println(max);                // prints 312
        }
}
```

Code starts executing inside main. First, we create a new Foo object, referenced via variable x. We invoke x's method named *maximum* with three parameters: 5, 312, and 55. This executes the code inside maximum and **binds** num1 with 5, num2 with 312, and num3 with 55.

The code inside *maximum* has logic to determine which number of the three parameters is smallest. This is done by defining a variable called curmax. Any variable defined inside a function is considered a local variable. It exists only within the scope of the function. When the function exits, this variable is gone forever! This is a good way to declare variables we need temporarily.

How do we get data back from the function to the caller? We can use the return statement, which returns any value we like back to the caller. In this case, we return the local variable curmax, and it is assigned into the variable max inside the main function before curmax is destroyed.

Here is another example of a method that converts a temperature specified in Fahrenheit to Celsius:

```
public double ConvertToCelsius(int tempInFahr)
{
        return ((tempInFahr – 32)*5.0/9.0);
}
```

Passing Primitive Parameters: Call ByValue

What we've done so far is pass the parameters using what is called call by value. For example we passed the variable num1 to a function. The function gets the value contained in the variable num1 and can work with it. However, any changes that are made to the variable are not reflected back in the calling program. For example:

```
class Foo {
        public void DoesntChange(int num1)
        {
                System.out.println(num1);               // Prints 1
                num1 = 5;
                System.out.println(num1);               // Prints 5
        }

        public static void main(String[] args) {
                Foo x = new Foo();
                int val = 1;
                x.DoesntChange(val);
                System.out.println("Back in main:" + val);              // Prints 1
        }
}
```

The output for this program is:

```
1
5
Back in main: 1
```

The variable "val" is unchanged in main. This is because the function DoesntChange treats its parameters like local variables. The contents of the variable from the caller is copied into the variable used inside the function. When the function exits, this variable is destroyed. The original variable retains its original value.

static Methods and Data Members

Static is another modifier that we can associate with methods or class variables. We indicate that something is static by inserting the keyword **static** before the method or variable.

An identifier that is static is associated with the class definition and not with an instance of the class. This is primarily used when we want to define a method that is completely self-contained and does not depend on any data within a class. However, Java requires that this method be defined inside a class. The solution is to put the method in a class, but declare it as static so that we don't have to create an instance of the class when we want to use it.

For example, consider the DoesntChange function we wrote earlier. To use this function, we had to create an instance of a class variable just so we could invoke the function:

```
Foo x = new Foo();
int val = 1;
x.DoesntChange(val);
```

The only purpose of created object x was to invoke the function. If we make this method a static method, then we don't need to create the object:

```
class Foo {
        public static void DoesntChange(int num1)
        {
                System.out.println(num1);               // Prints 1
                num1 = 5;
                System.out.println(num1);               // Prints 5
        }

        public static void main(String[] args) {
                int val = 1;
                Foo.DoesntChange(val);
                System.out.println("Back in main:" + val);           // Prints 1
        }
}
```

To invoke the function, we just give the name of the class followed by the function. We don't need to create an instance of the function to invoke it.

This would also be useful for our conversion function, ConvertToCelsius. We might make a class called Conversions that contains a number of static methods to perform conversions for us.

Class Example: Money Class

Let's add some more methods to the Money class to make it a bit more useful.

```
class Money {
        private int m_dollars;              // Most class variables are kept private
        private int m_cents;

        // Method to set the dollars and cents
        public void SetValue(int inDollars, int inCents)
        {
                m_dollars = inDollars;
                m_cents = inCents;
        }

        // Method to print the value
        public void PrintValue()
        {
                System.out.println(m_dollars + "." + m_cents);
        }

        // Method to get the dollar value          (Why do we need this?)
        public int GetDollars()
        {
                return m_dollars;
        }

        // Method to get the cents value           (Why do we need this?)
        public int GetCents()
        {
                return m_cents;
        }

        // Method to add another money value to this one
        public void AddMoney(Money mNew)
        {
                int newDollars, newCents;
                newDollars = mNew.m_dollars + m_dollars;      // Private access?
                newCents = mNew.m_cents + m_cents;

                // Increment dollars if we have more than 99 cents
                if (newCents > 99) {
                        newDollars = newDollars + (newCents / 100);
                        newCents = newCents % 100;
                }
                m_dollars = newDollars;
                m_cents = newCents;
        }
}
```

```
// This class uses the Money Class

class Test {
        public static void main(String[] argv) {
                Money m1 = new Money();
                Money m2 = new Money();

                m1.SetValue(300,21);        // Sets m1's dollars to 300, cents to 21
                m2.SetValue(2,99);          // Sets m2's dollars to 2, cents to 99
                m2.PrintValue();
                m2.AddMoney(m1);            // Adds m1's values to m2
                m2.PrintValue();
        }
}
```

The output is:

```
2.99            ← First initialized value for m2
303.20          ← New value for m2 with m1's value added to it
                  and normalized cents under 100
```

Notice the abstraction we have implemented in the AddMoney function. If we ever add together something more than 100 cents, then we automatically update the cents into dollars. This logic is hidden for us in the AddMoney() functionality of the Money class. If we had just let the user use normal addition, then the user would have to implement this logic themselves elsewhere.

There is much more to cover about classes, including inheritance, scoping of identifiers, exceptions, interfaces, and many other topics. However as we are out of time in CS101 these will have to wait for another day.