Classes and Methods

We have already been using classes in Java – a class corresponds to an object. Classes encourage 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, when we first started Java programming we described the HelloWorld class:

```
	public class HelloWorld
	{
		public static void main (String[] args)
		{
			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!

```
public class className
{
	// Define data members; i.e. variables associated with this class
	public/private datatype varname1;
	public/private datatype varname1;
	…

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

The term "data member" refers to a variable 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*.

Important: Java expects each class to be stored in a separate file. The name of the file should match the name of the class, with a ".java" appended to the end. For example, if you have two classes, one called `Main` and the other called `Money` then there should be two files, the first called `Main.java` and the second called `Money.java`.

Class Variables – Data Members i.e. Instance Variables

We already know what variables are. Variables defined inside a class are called *member variables,* because they are members of a class. They are also called *instance* variables because they are associated with instances of the class. Any code inside the class is able to access these variables.

Let's look at a simple class that contains only instance variables and no methods.

```
	public class Money
	{
		public int dollars;
		public int cents;
	}
```

Now consider another class that creates objects of type Money:

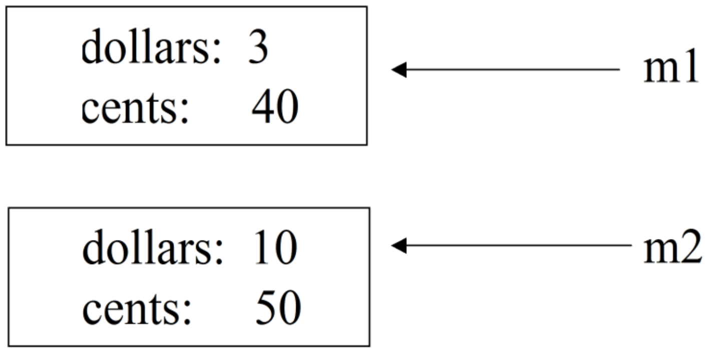
```
public class Test
{
  public static void main(String[] args)
  {
      Money m1 = new Money();
      Money m2 = new Money();

      m1.dollars = 3;
      m1.cents = 40;
      m2.dollars = 10;
      m2.cents = 50;
      System.out.println(m1.dollars + " " + m1.cents);
      System.out.println(m2.dollars + " " + m2.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:

```
public class Employee
{
      public String name;
      public int age;
      public double hourlyWage;
      public long idNumber;
}
```

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

Class Methods

We can also write methods in a class. When we don't use the keyword static, these are non-static methods. This is the normal way of writing a method, so usually when we say "method" we mean a non-static method. A non-static method is one that applies only to the class it is in. It will generally operate on a data variable defined in the class. If it doesn't do something with a data variable then it might make more sense to make it a static method instead of non-static.

To define a method use the normal template but omit the keyword static:

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

For example, for our Money class, here is a method named print that outputs the value of the money object:

```
public class Money
{
      public int dollars;
      public int cents;
      public void print()
      {
            System.out.println("Dollars: " + dollars +
                                "Cents: " + cents);
      }
}
```

In main:

```
      Money m1 = new Money();
      m1.dollars = 3;
      m1.cents = 40;
      m1.print();
```

Controlling Access to Instance Variables or Methods

As we saw with the money example, by default we have access from outside the class to any variables we define. We accessed "dollars" and "cents" from outside the Money class, when we were in the Test class. 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:

```
public class Money
{
      public int dollars;
      private int cents;
}

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

            m1.dollars = 3;     // VALID, dollars is public
            m1.cents = 40;      // INVALID, 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 or private 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 variables defined in the class, not to variables defined inside a method (e.g., we won't use private/public on variables defined inside main).

Encapsulation

If we make instance variables **private** then those variables can only be accessed from inside the class. If we want to make these variables accessible to the outside world then the recommended approach is to add **public** methods that get or set those private variables.

Why? This allows us to "hide" the internal workings of the variables and make a public "interface" that allows us to write code that ensures the variables are used correctly. In our money example, if we want to set the dollars or cents variable then we can do so by making a public method that sets or gets those variables. These are called **accessors** (when getting the variable) or **mutators** (when setting the variable). For example here is an accessor and mutator for the dollars variable:

```
private int dollars;

// Accessor for dollars
public int getDollars()
{
      return dollars;
}


// Mutator for dollars
public void setDollars(int newDollars)
{
      dollars = newDollars;
}
```

This might seem like it is extra work. It's simpler to just make the dollars variable public so it can be accessed from outside the class. But without it, someone could do potentially illegal things like:

```
money1.dollars = -100;
```

If we wanted to prevent this we could make the `dollars` variable private so it's not directly accessible, then write the `setDollars` method like so:

```
// Mutator for dollars
public void setDollars(int newDollars)
{
      if (dollars >= 0)
            dollars = newDollars;
}
```

Another benefit of encapsulation is that by forcing any code outside the class to call our accessor/mutator methods, we are now free to change how the class is implemented but not have to modify the code that uses our class.

For example, let's say that we decide to change the dollars and cents in the Money class from two integers to a single variable of type double. For example, maybe we decide we need to represent amounts less than one cent. If the dollars and cents variables were public, the code would break that tries to directly access these variables.

But through encapsulation we can make the appropriate conversions in the accessors and mutators such that any code outside the class will still work. This idea is shown below:

```
// Money class with accessors/mutators for the dollars and cents
public class Money
{
      private int dollars = 0;
      private int cents = 0;
```

```
	public void setDollars(int newdollars)
	{
		dollars = newdollars;
	}

	public int getDollars()
	{
		return dollars;
	}

	public void setCents(int newcents)
	{
		cents = newcents;
	}

	public int getCents()
	{
		return cents;
	}

	public static void main(String[] args)
	{
		Money m1 = new Money();

		m1.setDollars(10);
		m1.setCents(25);

		System.out.println(m1.getDollars() + " " +
		                   m1.getCents());
	}
}
```

Here we changed the implementation to use a double instead. Note that no code inside main needs to change (although there is some messy logic inside the accessors/mutators).

```
public class Money
{
	private double amount = 0;

	public void setDollars(int newdollars)
	{
		double cents = amount - (int) amount;
		amount = newdollars + cents;
	}

	public int getDollars()
	{
		return (int) amount;
	}
```

```
      public void setCents(int newcents)
      {
            int dollars = (int) amount;
            amount = dollars + (newcents / 100.0);
      }

      public int getCents()
      {
            double cents = amount - (int) amount;
            // avoid roundoff errors, e.g. 0.009999 cents
            return (int) Math.round(cents * 100);
      }

      public static void main(String[] args)
      {
            Money m1 = new Money();

            m1.setDollars(10);
            m1.setCents(29);
            System.out.println(m1.getDollars() + " " +
                                    m1.getCents());
      }
}
```

Class Constructors

Because we use classes to encapsulate data, it is essential that class objects be initialized properly. When we defined the Money class, we were relying upon the user to set the value for dollars and cents outside the class. What if the client forgets to initialize the values? This can be such a serious problem that Java provides a mechanism to guarantee that all class instances are properly initialized called the class constructor.

A class constructor is a method with the same name as the class and no return type. We can even make multiple constructors with multiple parameters, to differentiate different ways a class may be initialized.

The constructor with no parameters is called the **default constructor and it is highly recommended that you always create one.**

Below are two constructors for the Money class along with a method to print the currency value:

```
public class Money
{
      private int dollars;    // Most class variables are kept private
      private int cents;

      // This constructor invoked if we create the object with no parms
      public Money()
      {
```

```
		dollars = 1;
		cents = 0;
	}

	// This constructor invoked if we create the object with
	// a dollar and cent value
	public Money(int inDollars, int inCents)
	{
		dollars = inDollars;
		cents = inCents;
	}

	// Method to print the value
	public void printValue()
	{
		System.out.println(dollars + "." + cents);
	}
}

public class Test
{
	public static void main(String[] argv)
	{
		Money m1 = new Money();
		Money m2 = new Money(5, 50);
		m1.printValue();
		m2.printValue();
	}
}
```

When this program runs, the output is:

```
	1.0			← From m1
	5.50			← From m2
```

When we create m1, we give no parameters in Money(). This invokes the default constructor, which is given as:

public Money()

This code initializes dollars to 1 and cents to 0.

When we create m2, we give two parameters in Money(5,50). Java will then search for a constructor that has two parameters that match the ones provided. The constructor that is found is then invoked:

public Money(int inDollars, int inCents)

This code then sets the member variables to the input parameters, resulting in the output previously specified.

Class Example: Money Class

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

```
public class Money
{
      private int dollars; // Most class variables are kept private
      private int cents;

      // Constructors
      public Money()
      {
            dollars = 1;
            cents = 0;
      }
      public Money(int inDollars, int inCents)
      {
            dollars = inDollars;
            cents = inCents;
      }

      // Method to print the value
      public void printValue()
      {
            System.out.println(dollars + "." + cents);
      }

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

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

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

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

```
public class Test
{
	public static void main(String[] argv)
	{
		Money m1 = new Money(300,21);
		Money m2 = new Money(2, 99);

		m2.printValue();
		m2.addMoney(m1);
		m2.printValue();
	}
}
```

The output is:

```
	2.99			← First initialized value for m2
	303.20
```

Notice the abstraction we have implemented in the AddMoney method. 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.

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:

```
public 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.

Some commonly used static methods are in the Math, Integer, Double, and other wrapper classes.

If we put static in front of an instance variable, then there is only one copy of that variable shared among all instances of the class. We'll have more to say about static variables later.

A common error that is made is trying to invoke a non-static method from a static method. **A static method can't call a non-static method unless there is a calling object that contains the non-static method. Similarly a static method can't access any non-static variables without a calling object.**

However, non-static methods can access static methods or variables.