Classes and Methods

We have already been using classes in Java – as we have seen, 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;
}
```

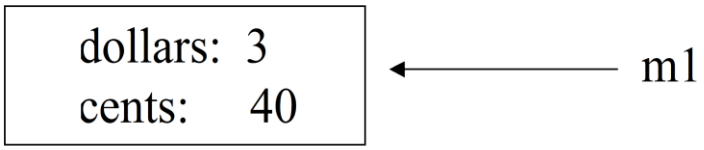
Now consider another class that creates objects of type Money:

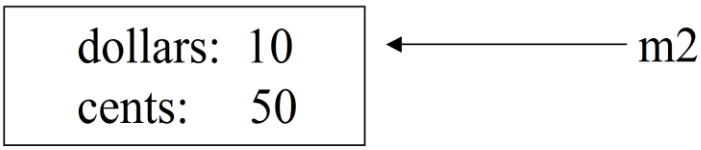
```
public class Test
{
  public static void main(String[] args)
  {
      Money m1 = new Money();
      Money m2 = new Money();

      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.

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).

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 methods 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 method to make sure it works properly)

Let's examine how to write programs using methods.

Before starting

We've already been using quite a few methods in our programs. Calls like println() and nextInt() 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 methods 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 methods, it is a good idea to look at the problem you are trying to address and see how it might logically be broken up into pieces. For example, if you are writing a program to give a quiz then you might have separate methods to:

- Load the questions and answers from the disk
- Ask a single question to the player and get an answer
- Test if an answer is correct
- Display the final score
- Coordinate all of the actions above (might go in the main method)

Defining a Method

To define a method use the following template:

```
modifier return_type methodName(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. (There is also a modifier *protected* but we will defer its discussion for now).

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.

methodName is an identifier selected by the user as the name for the method.

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:

```
public 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 method 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);
}
```

Filling in the code using Top-Down Design

Two ways of completing our program are to use either top-down or bottom-up design. In top-down design, we write the most general code first and then fill in the details later. In bottom-up design, we write the detailed code first and then incorporate them into more general code. In both cases, testing is done along the way. This is called incremental testing. It is generally a bad idea to write a lot of code and assume that it will run – this approach is very hard to debug if there are errors.

Let's first look at top-down design. Generally this means we start by defining main. Here is an example of a dice game:

```
public static void main(String[] args)
{
      int wallet = 1000;
      boolean keepPlaying = true;
      int roll, bet;
      DiceGame dg = new DiceGame(); // New DiceGame object

      while (keepPlaying)
      {
            bet = dg.GetBet(Wallet);
            roll = dg.RollDice();
            System.out.println("Roll=" + roll + " bet=" + bet);
            // Other stuff here to process the bet and the roll
      }
}
```

This makes for a nice and simple main method! This method assumes we have another class, called DiceGame, that has methods associated with it to perform the betting and dice rolling activities. However, we haven't yet filled in what GetBet() and RollDice() will do.

If you wish to test what you've created so far, a good practice is to write **stubs** for each function. The stubs simply print out dummy information, so that you can test to see if the calling code is functioning properly. Here are some sample stubs:

```
class DiceGame
{
      public int RollDice()
      {
            // Define roll dice here
            return (5);// always returns the number 5
      }

      public int GetBet(int cashAvailable)
      {
            int bet;
            Scanner keyboard = new Scanner(System.in);

            System.out.println("Enter bet:");
            bet = keyboard.nextInt();
            // More GetBet code here, need to validate input still
            // To make sure it is a valid amount
            return (bet);
      }
}
```

Note that the GetBet method has access to the parameter named "cashAvailable" , which holds the value that was passed into from the calling routine. This is referred to as a local variable, and will be described more later. There is also a local variable called "bet".

This program will now compile and run, and give the output:

```
Enter bet:
4
Roll is 5 bet is 4
Enter bet:
3
Roll is 5 bet is 3
```

This allows us to test to see if the outer loop is working properly. If there were bugs, like the wrong value was being returned, or other problems, then you could hopefully fix them before moving on!

The next step would be to flesh out the rest of the main function and have it handle the other cases for the game. Once again, these cases can be tested using the stubs that we wrote for GetBet and RollDice.

After main has been written and debugged, the next step is to flesh out the individual functions. If GetBet() invoked functions of its own, we could follow the same top-down

procedure and write the code for GetBet(), leaving stubs for functions called by GetBet. However since GetBet is pretty short without calling any functions we can just write it all:

```
public int GetBet(int cashAvailable)
{
     int bet;
     Scanner keyboard = new Scanner(System.in);

     do {
          System.out.println("Enter bet:");
          bet = keyboard.nextInt();
     } while ((bet > cashAvailable) || (bet <= 0));
     return (bet);
}
```

We can do the same for RollDice:

```
public int RollDice()
{
          return ((int) (Math.random() * 6)  +
                   (int) (Math.random() * 6) + 2);
}
```

When we put this all together, the entire program is finished!

Filling in the code using Bottom-Up design

We can take the opposite approach and write our functions using a bottom-up process. In this case, we would write the individual, detailed functions first, and the more general code later. For example, let's say that we wrote both the RollDice and GetBet methods, but we haven't written main() yet. The process to follow is to test each individual function using some type of test harness. Once we're satisfied that the individual functions work, we can move on and start writing the main function.

```
// Test harness main to see if RollDice is really working
public static void main(String[] args)
{
      int i;
      dg = new DiceGame();
      for (i=0; i<100; i++)
      {
            System.out.println(cg.RollDice());
            // Expect 100 random nums
      }
}
```

In this case, our tests might discover that RollDice returns an invalid dice roll, like a total of 1 or 0, or perhaps gives the wrong distribution of numbers. Depending on what you want to do (print an error? Print nothing?) you might want to go back and change the function accordingly.

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:

```
public class Foo
{
      public void doesntChange(int num1)
      {
            int y = 3;            // Local variable, used later
            System.out.println(num1);        // Prints 1
            num1 = 5;
            System.out.println(num1);        // Prints 5
            return;
      }

      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.

Passing Object Parameters: Call By Reference

We get a different behavior if we pass an object as a parameter rather than a primitive data type. If we change the **contents – i.e. a member variable** of an object that is passed as a parameter inside a function, then the change is reflected back in the calling code. This is called call by reference. Setting the object itself to something different does not reflect back in the calling code. For example, consider the Money class we defined earlier:

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

public class Foo
{
 public void doesChange(Money cash)
 {
      System.out.println(cash.dollars);         // Prints 1
      cash.dollars = 5;
      System.out.println(cash.dollars);         // Prints 5
 }

 public static void main(String[] args)
 {
      Foo x = new Foo();
      Money dinero = new Money();
      dinero.dollars = 1;
      x.doesChange(dinero);
      System.out.println("Back in main:" + dinero.dollars);
      //Prints 5
 }
}
```

The output is:

1
5
Back in main: 5

The variable is changed back in the caller because we're passing an object as a parameter. Internally, objects are passed to the function by providing the address of the object in memory. This means changes to the object inside the function change the original copy

of the data. This is not the case when we pass primitive data types; instead we give a copy of the entire variable instead of providing the address.

Underneath the Hood – The Stack and Parameter Passing

What is happening when we call functions and pass parameters back and forth? It is very useful to know how this is done architecturally in the computer. Local variables and parameters are passed using the stack in the computer.

The Stack

You can visualize the stack like a stack of trays in a cafeteria. A person grabs a tray from the top of the stack. Clean trays are also added to the top of the stack. In general operation, you never touch anything that is not on top of the stack. This has the property of LIFO – Last In, First Out. The last thing you put into the stack is the first thing that will be taken out. In contrast, the first thing put into the stack is the last thing that is taken out.

The computer implements the stack using a chunk of memory, with a stack pointer that points to the top of the stack. Let's say our stack pointer is currently pointing to memory address 255, and we want to add the number 3 to the stack To do so, we decrement the stack pointer and add the value:

Before:
Stack Pointer: 255

Memory Address	Contents
0	?
1	?
…	?
254	?
255	?

After "pushing" the number 3 on the stack, we then decrement the stack pointer so it points to the top:

Stack pointer: 254

Memory Address	Contents
0	?
1	?
…	?
253	?
254	3
255	?

Now let's say we push the number 4 on the stack:

Stack pointer: 253

Memory Address	Contents
0	?
1	?
…	?
253	4
254	3
255	?

If we now "pop" the value from the stack, we would retrieve back the number 4, and increment the stack pointer by 1, giving:

Stack pointer: 254

Memory Address	Contents
0	?
1	?
…	?
253	?
254	3
255	?

We could continue by popping off the 3 and returning to our original state.
Note that the computer must know where the bounds of the stack are, or we might overrun into memory that is not allocated for the stack.

What is the stack used for? When we call a method, it is used to store the contents of variables and to remember what the next instruction is that we should execute. This is needed in case there is an arbitrary number of nested methods.

Let's see how our stack might look upon calling the DoesntChange example given in Call By Value. Variables are actually defined in the stack, so when main executes, it is going to declare variables x and val on stack. These are called **automatic** or **auto** variables.

Upon completion of variable initialization in main:

Stack pointer: 254

Memory Address	Contents	
0	?	
1	?	
…	?	
253	?	
254	1	// val's storage in main
255	(address of a Foo object)	// x's storage in main

When main is executing, the computer has to remember that 255 on the stack corresponds to the bottom, and 254 corresponds to the top. Variables within this range are local variables that main can access.

What happens when we make the method call to DoesntChange() ? First, there is a parameter *val* that is passed into the function. The function uses the variable name num1. This is a separate variable than the value in main, so a new variable is defined by allocating space for it on the stack. This variable is pushed onto the stack and we decrement the stack pointer.

Stack pointer: 253

Memory Address	Contents	
0	?	
1	?	
…	?	
253	1	// num1's storage, set to val
254	1	// val's storage in main
255	(address of a Foo object)	// x's storage in main

Next, we need to remember where to resume execution when the ChangeValues function exits. So the next thing that gets pushed onto the stack is the address of the next instruction in main. In this case, the address corresponds to the instruction that begins with

```
System.out.println("Back in main: " + val);
```

Let's say that this address is 9999. The stack now looks like:

Stack pointer: 252

Memory Address	Contents	
0	?	
1	?	
…	?	
252	9999	// Address of next instruction in main
253	1	// num1's storage, set to val
254	1	// val's storage in main
255	(address of a Foo object)	// x's storage in main

Now, note that we declare a local variable, Y, in DoesntChange. Once again, we will say more about local variables later. But for now, this is a new variable that we have to define. We define it by pushing it on the stack:

Stack pointer: 251

Memory Address	Contents	
0	?	
1	?	
…	?	
251	3	// Y's storage in DoesntChange
252	9999	// Address of next instruction in main
253	1	// num1's storage
254	1	// val's storage in main
255	(address of a Foo object)	// x's storage in main

When we execute the statement "num1=5;" the place in memory that gets changed is the corresponding location on the stack that has been allocated for num1. Note that the location where we are storing x in main is unchanged.

Stack pointer: 251

Memory Address	Contents	
0	?	
1	?	
…	?	
251	3	// Y's storage in DoesntChange
252	9999	// Address of next instruction in main
253	5	// num1's storage
254	1	// val's storage in main
255	(address of a Foo object)	// x's storage in main

When the return function is executed, we pop off the values in the stack all the way back to the place where the return pointer is stored. This has the effect of throwing away the local variables. Some computers will also push the return value on the stack (in this case 10), while others will pass the return value back to the calling program by placing the return value inside one of the registers of the CPU (a place for fast temporary storage).

Upon popping off 9999 we store that value into the Instruction Pointer so that the next instruction we execute begins at memory address 9999. We also need to pop off the storage parameters so that we are back to the original state of the stack before making the function call:

Stack pointer: 254

Memory Address	Contents	
0	?	
1	?	
…	?	
251	3	// Y's storage in DoesntChange
252	9999	// Address of next instruction in main
253	5	// num1's storage
254	1	// val's storage in main
255	(address of a Foo object)	// x's storage in main

Note that the old values are still sitting on the stack, but by changing the stack pointer back to 254, any new items pushed on the stack will overwrite the old values.

The process of using the stack varies from machine to machine and compiler to compiler, but the basic process is the same. Note that each time we call a function, we'll have to allocate some memory off the stack – this will have implications later when we do recursion and make many function calls. If we make too many calls, we will exhaust the amount of available memory on the stack.

How does call by reference work? Briefly, instead of copying the value of a primitive type into the stack, instead a pointer containing the memory address of an object is put on the stack. When accessing this location the computer ends up making changes to the one and only copy of the data, which happens to be referred to by the class variable.

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.