Static Methods

A **method** is just a collection of code. They are also called **functions** or **procedures**. It provides a way to break a larger program up into smaller, reusable chunks. This also has the benefit of making programs easier to understand. You are already familiar with one method, the **main** method. 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***.

For now, we will only study a type of method called a **static method**. With a static method, there is only one of these for the whole program – you can think of it like a globally accessible method. Later we will examine methods that don't use the static keyword.

Why use methods?

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 method many times in one program)
- Promotes software reuse (reuse a method in another program)
- Promotes good design practices (Specify 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 static 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). For now we'll use **public**.

Static may be left off. If it is included then there is only one of these methods that can be thought of as a global method accessible to anyone, without the need for a calling object. When static is left off, the method can only be called from certain places. For now, we'll use **static**.

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 static 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)
      {
            int max;

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

Code starts executing inside main. We invoke the 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 static 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;

      while (keepPlaying)
      {
            bet = getBet(Wallet);
            roll = 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! Main assumes that there are 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 static int rollDice()
      {
            // Define roll dice here
            return (5);// always returns the number 5
      }

      public static 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 methods 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 static 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 static int rollDice()
{
     Random rnd = new Random(); // Better to define elsewhere
                                // but OK here
     return (rnd.nextInt(6)  +
                  rnd.nextInt(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;
      for (i=0; i<100; i++)
      {
            System.out.println(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 By Value

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 static 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)
      {
            int val = 1;
            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. We create objects with the keyword **new**. We haven't defined our own objects yet, but an array is an example of an object we have seen already.

If we change the **contents – e.g. for an array, this would be the data inside the array -** of an object that is passed as a parameter inside a method, 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:

```
public class Foo
{
 public static void doesChange(int[] arr)
 {
      arr[0] = 100;
 }

 public static void main(String[] args)
 {
      int[] a = new int[3];
      a[0] = 1;
      doesChange(a);
      System.out.println("Back in main:" + a[0]);     //Prints 100
 }
}
```

First, some mechanics of how to pass an array. **When we define the method, the array parameter is defined using [] after the data type followed by a parameter name**, just like when it is defined in main. We don't need to specify the array size.

When we send an array into a method, we give the variable name but no square brackets. Just the variable name by itself.

The variable is changed back in the caller because we're passing an object as a parameter. Internally, objects are passed to the method 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

We will give a discussion of this in class. There is a special area of memory called the stack. Every time we call a method, we create a stack frame on top of the stack that holds any variables allocated for the method.