Computer Science 523
Advanced Programming
The College of Saint Rose
Summer 2014

Topic Notes: Java Overview/Review

There are many ways to write a program to solve a particular problem correctly. In addition to correctness, we will think about these implmentation goals for our programs:

1. Efficiency
    - use algorithms, data structures, and language constructs to mimimize the amount of processing power and memory needed
2. Robustness
    - produce correct output for all inputs - including erroneous input
3. Adaptability
    - a program can evolve over time with new requirements
4. Reusability
    - develop general-purpose code that may be used in multiple situations

---

Programming Languages

We will be programming in Java.

There are many programming languages, and Java is just one example. Most programming languages all basically do the same thing, though some are much better than others at certain types of tasks. We choose to study advanced programming in Java because it is a modern, object-oriented language that runs on all modern computer systems. An appropriate choice of programming language makes it easier to write high-quality software.

All computer languages are an *abstraction* to make it more convenient to get computers to do what we want them to do. We could write in 0's and 1's, but a variety of languages have been developed to facilitate the development of software.

Most langauages, including C and C++ have compilers that translate source code into a executable program that runs on a particular machine. Java and some other modern languages are different. All Java compilers translate to a particular *virtual machine*, which, in turn, runs on specific computers.

This gives Java some advantages that we will discuss along the way. For now, we will look at programming in Java as a general-purpose modern object-oriented programming language.

Java Basics

We begin with the obligitory first example:

See Example: Hello

Things to note in Hello.java:

- Everything in Java has to be in a *class*. More about classes in a minute. In C, there are no classes and in C++ there are classes, but you can write functions outside of any class in addition to class methods.
- Some of you may have seen only Java *applets* – this is a Java *application*. We will look at both during this semester.
- Each class can have *methods*. An application has a class that must have a `main` method with the method signature:

  ```
  public static void main(String[] args)
  ```

  Exactly what is meant by all of these will become clear later, but basically this is where execution will start when we run this program.
- To execute this program in BlueJ, we open the project, and click on the class or classes to bring up the source code. Compile with the "Compile" button (obviously). To run, we go back to the project window, right click on the class that contains the `main` method we want to execute, and choose the entry from the dropdown to execute the `main` method.

  A console window should pop up with your output. If your program was taking input, you would use that same window.
- Comments:

  ```
  // starts comments which go to end of line
  /* multi-line comments
        are done like this */
  ```

  You most certainly have been strongly encouraged or, more likely, required to document your program using *comments* in your previous courses.

  The comment above the header in `Hello.java` is a special kind of comment, it starts with `/**` indicating that it is a *Javadoc* comment. These comments are used to generate documentation for Java programs automatically. We may look at Javadoc more later.

- The text output is made by a call to `System.out.println()` which takes one argument – a string to print to the terminal.

  This plays the role of C's `printf` and C++'s `cout`.

  Note: `System.out.print()` does the same, but without the new line at the end.

We can extend this just a bit to see more about how to produce useful output.

See Example: Seuss

This is basically `HelloWorld` all over, but there are a few little items that are new.

- Notice that the sentence "We know how." is printed on the same line as "Well, we can do it.". That's because we used a different printing method for the latter: `System.out.print`. This one works the same as `System.out.println` except that the output is not advanced to the next line at the end.
- The last statement includes some *escape sequences* that cause the output to be formatted a bit differently than it would otherwise appear. Escape sequences begin with a `\` character and are followed by a *control character* that defines the behavior of the sequence. Here, we have three:

  1. `\t` inserts a "tab" character, effectively indenting our output in this case,
  2. `\n` advances the output to a new line, and
  3. `\"` prints the double quote character, which would otherwise be impossible since a regular `"` character would be interpreted as the end of the text we are trying to print.

One bit of terminology at this point: the methods `System.out.println` and `System.out.print` are part of the *Java API* (Application Programmer Interface). Any valid Java installation comes equipped with an extensive collection of pre-written software that our programs can use.

Interactive Programs

To create nearly any interesting program, we need to be able to provide it with *input*. This will allow the program to react differently when presented with different inputs.

See Example: HelloYou

First, to get information from the keyboard into our program, we again turn to the Java API. There are several mechanisms available, a few of which we will see this semester. But we will start with one called a `Scanner`.

In order to use a `Scanner`, we will need to tell Java that we intend to use it, by inserting the line:

```
import java.util.Scanner;
```

at the top of our program (before the class header). We will see later how to determine exactly what to "import" in these *import statements* to use various Java API functionality, but for now, just know that this is what we need to do to use a `Scanner`.

Then, in our `main` method, where we wish to access information from the keyboard, we *construct* a `Scanner` that we can use in our program. This is done with the line:

```
Scanner input = new Scanner(System.in);
```

This is the first example we've seen here of an *object construction*. There's actually quite a bit going on in this line, and we'll examine it more carefully in a minute. But for now, we're creating a `Scanner` that uses `System.in` (which is Java's cryptic way of saying "what is typed at the keyboard"), and giving it a name, `input`. `input` is a *local variable*, which is a fundamental construct in nearly any programming language, and again one that we will examine more carefully in a moment.

Now that we have a `Scanner` called `input`, we can ask it to give us the next chunk of text that was typed at the keyboard. There are many possibilities for what we mean by "chunk" but for now, we just want the word that someone types in as their name. Java's `Scanner` provides a method that does just that, called `next`.

We will also need a name for the word that was typed in, so we can print it out later. This is all accomplished with the line:

```
String name = input.next();
```

Before we carry on further, we need to consider the concept of variables a little more closely.

A *variable* is a named storage location in the computer's memory. We use a variable when we have determined that there is some piece of data that we have in one Java statement, and we need to remember that information for use in later statements.

When we need a variable in our program, we must *declare* that variable to Java, which is just a fancy way of saying that we are going to introduce a name to our Java program and tell Java what *type*, or kind, of data we intend to store there.

A variable declaration takes one of two forms:

```
type name;
```

or

```
type name = initialValue;
```

where "type" is the *data type* (or " kind of data") we will store, and "name" is the identifier we intend to use to refer to that data. In the second form, we also *initialize* the variable to have a specific value.

You have certainly seen many data types that store a variety of kinds of information. For now, we are using two of Java's:

- `Scanner` – which is the keyboard input mechanism we wish to use
- `String` – a collection of text, like a word or sentence

We give our `Scanner` the name `input` and the `String` the name `name`.

When naming our variables, we need to keep in mind several considerations:

- The name must be a valid Java *identifier*. This means it must consist only of letters, numbers, the dollar sign character, and the underscore character (though it can only start with a letter).
- The name should follow Java's *naming conventions*. Recall that for variables, we use lower-case letters, except when we have a name that is made up of multiple words, in which case we capitalize all but the first word.
- The name should be meaningful. That is, it should give some indication of what the variable is to be used for. The names here satisfy that requirement: `input` implies that this is where we get our input, and `name` implies that this is the name of something.

Once we have a variable, we can make use of its value later in our program. We do that here when we call the `next` method of our `Scanner` named `input` and when we use the `String` named `name` in the `System.out.println` statement at the end of our program.

One last new idea here is that we now have something more complex as the text to be printed by `System.out.println`. It's not just some text in double quotes, but some text in double quotes, followed by a `+`, followed by the name of our `String` variable.

This is an example of *string concatenation*. We have the *string literal* (*i.e.,* some text inside double quotes) to which we "append" the text in the variable `name`.

Working With Numbers

You know that computers often do just that: they compute with numbers. So next, we consider some examples of programs that work with numbers to show or remind you how Java works with numeric values.

Integer Values

We start simple. Let's compute a rectangle's area and perimeter.

See Example: Rectangle

There are a few things to note in this program.

First, we are working with numbers rather than words. This changes how we read the data from the keyboard through our `Scanner` and the type of variable we need to declare to store that data.

For this example, we are requiring that the dimensions of the rectangle are integer values.

The Java type we will most often use to store an integer value is an `int`. We declare and initialize `int` variables named `width` and `height` to store the rectangle's dimensions.

`int` is one of Java's *primitive data types*. We will see several other examples. These are the only types that are usually specified with an all-lowercase keyword.

We next need to use a different method of `Scanner` to force it to look for an integer and return it in as a Java `int` instead of a `String`. That method is called `nextInt`.

Once we have our width and height, we need to compute the area and perimeter from them. For this, we need to declare two more `int` variables and perform some computation to compute their values.

If you remember your elementary school geometry, you know that to compute the area of a rectangle, we multiply its width by its height. And to compute the perimeter we add up the lengths of all sides, which in this case is twice the width plus twice the height.

Java uses a notation to specify mathematical computations (a *mathematical expression*) that is mostly familiar from math. As we can see from the statement that computes `area`, we use the `*` *arithmetic operator* to specify multiplication.

So that statement instructs Java to multiply together the `int` value found in the variable `width` by the `int` value found in the variable `height` and store the product in the `int` variable `area`.

The computation of `perimeter` is a bit more complicated, but still pretty straightforward. We see that addition is specified by `+` and that we can use numbers in our expressions as well as values stored in variables.

We do need to know in what order Java will perform the operations here. If it does `2 * width`, then adds 2 to that result, multiplying that resuly by `height`, we will get the wrong answer. Fortunately, Java follows a strict *order of operations*. In this case, we say that multiplication has a higher *precedence* than addition, so Java will compute `2 * width`, then `2 * height`, then add together those results.

We will look in more detail at order of operations as we encounter other mathematical operators in subsequent examples.

Finally, we print out our results. We can see here that Java "does the right thing" when we concatenate string literals with `int` values.

Floating-point Values

Of course, are numbers are not always integers. Our next example, is to perform a simple miles per

gallon computation. Again, we will prompt for inputs, compute our answer, and report the result.

What do we need to know to make this example work?

- If we want to store non-integer values, which are called *floating-point values* in Java, we use variables of type `double` instead of `int`.
- If we want to read in `double` value from a `Scanner`, we use the `nextDouble` method.
- Division is specified by the `/` operator.

See Example: MilesPerGallon

Note the difference between integer division and floating-point division by trying the above first with `int` data, then with `double` data.

When we divide two `int` values using `/`, the result is the *quotient*, and we throw away the remainder. If we want the remainder (and only the remainder), we can use the `%` operator, often called the "mod" operator as it performs modulo arithmetic.

Any division operator where both operands are `int` values, results in an `int` quotient. If *either* operand (or both) is already a `double`, the results is a `double` and the answer would include any fractional part as a decimal.

Operator Precedence

We can specify complex arithmetic expressions using any combination of the following:

`*`	multiplication
`/`	division
`%`	remainder
`+`	addition
`-`	subtraction

In a long expression such as

```
12 + 9 / 4 - 18 % 4 * 19
```

there are choices to be made in how to evalulate. Fortunately, Java makes these decisions and makes it clear to us how it will evaluate such an expression.

1. *unary* negation operators are applied first, working left to right if there are multiple such operations
2. multiplications, divisions, and remainders are computed, again left to right

3. additions and subtractions are computed, left to right

So in the above expression, we first check for unary negations, and there are none.

Then, we do the multiplication, division, and remainder operations. Since these are all integer values, the any division will be computed as an integral quotient.

So, the `9 / 4` evaulates to `2` first. Giving

```
12 + 2 - 18 % 4 * 19
```

Next, `18 % 4` is evaluated to `2` (the remainder when we divide 18 by 4). Giving:

```
12 + 2 - 2 * 19
```

One multiplication remains, so we compute the `2 * 19` as `38`, giving:

```
12 + 2 - 38
```

We are left with only additions and subtractions, which are evaulated left to right. `12 + 2` becomes `14`, leaving us:

```
14 - 38
```

and after the last subtraction, we have `-24` for a final result.

The same rules apply if we have data in variables declared as either `int` or `double` values.

If we wish to override the default rules, just like in math, we can place parentheses around any lower-precedence operation that we wish to have performed before some higher-precedence operation, or if we want to change the order among same-precedence operations to do some further to the right before some further to the left.

Named Constants

The next program, which we will develop in class is going to do the following:

- Read in 2 lines of input. Each contains the name of a baseball team (which must be a single word) and the number of runs that team scored.
- Report the total runs scored.
- Report the average number of runs per inning, both as a decimal and as a mixed number (a whole number followed by a fraction).

See Example: RunsScored

This example is the first one that demonstrates an important feature of good programming style: the use of *named constants*.

In this case, we are going to assume a baseball game has 9 innings. But perhaps in some other cases, there are 6 or 7 inning games. So we define a constant:

```
final int NUMBER_OF_INNINGS = 9;
```

As our programs become more complex, we will be using many numeric values. A *literal* is a value that is written into the code of a program that is not stored in a variable. Using many somewhat arbitrary numeric values as literals in a program can make the program difficult to understand and modify. We can improve the situation by associating the values with names so that we are reminded what the values signify when we see the names used.

Java includes a mechanism to enable us to use such names effectively. If you include the word "`final`" in a variable's declaration, this indicates that the value assigned to it in the declaration will never change. This means that its value cannot be changed (possibly by mistake).

It also means that we can change the value in just that one place if we decide to change the number of innings in a game (at which point we would have to recompile our program). Otherwise, we'd need to remember to change all instances of the number if we changed any.

A couple other notes from this program:

- We read both a `String` and an `int` from the same keyboard input line.
- We need to be careful that the addition of the two scores are done before the string concatenation in our printout.
- We need to be careful that the result of our `int` division does not throw away any fractional part in the case where we want to store the result in a `double`.
- We use both integer division and the remainder operator to compute our mixed number result.

Conditionals

The few programs we have seen so far have had one thing in common: they are entirely *sequential*. The statements in our `main` methods all execute, one after another, in the order they are encountered in our program.

From your programming experience, you know very well that programs need to be able to *make choices*. If a certain condition is "true" we would like to do one thing, if it's "false" we would do something else or possibly nothing at all.

Thinking of every day algorithms, this is something we do all the time.

- If I am still hungry, I will go for seconds.
- If it is a weekday, I will set my alarm to get up for class. Otherwise, I will sleep as long as I would like.
- If the dough is too watery, add more flour.
- If the student's score is at least 95% on the spelling test, put a sticker at the top.

Java's `if` Statement

The workhorse of Java (and most programming languages) for *conditional execution* is the `if` *statement*.

The basic form is

```
if (<boolean condition>) {
   statement1;
   statement2;
   ...
}
```

where `<boolean condition>` is some Java expression that evaluates to `true` or `false`. If it evaluates to `true`, the statements inside the curly braces following the condition are executed. Otherwise, they are skipped.

Used in a program:

See Example: OlderThanJava

There are many things that can be used to construct a boolean expression, but we will start with the standard relational operators and use them to compare numeric values.

Expression	When is it `true`?
`x > y`	when `x` is greater than `y`
`x < y`	when `x` is less than `y`
`x >= y`	when `x` is greater than or equal to `y`
`x <= y`	when `x` is less than or equal to `y`
`x == y`	when `x` is equal to `y`
`x != y`	when `x` is not equal to `y`

Java's `if-else` Construct

The `if` statement we saw above allows us to execute a statement or group of statements if the condition is true. Often, we want to execute one set of statements if the condition is true and another set if the condition is false.

We will expand our `OlderThanJava` program to do this. Here, we print a different message if the person is younger than or the same age as Java (in addition to the previous message when the person is older).

In Java, we can use the `if-else` construct.

```
if (<boolean condition>) {
   statement1A;
   statement1B;
   ...
}
else {
   statement2A;
   statement2B;
   ...
}
```

where `<boolean condition>` is some Java expression that evaluates to `true` or `false`.

See Example: OlderYoungerThanJava

We will next consider another example of `if` statements in Java, but with the added bonus of using a different mechanism for input and output.

The problem: we wish to write a program that calculates the number of full payments needed for a no-interest loan where we are given a loan amount and desired monthly payment. This number is reported. If additional funds are due after those full payments are made, that is reported as well.

See Example: NoInterestLoan

The comments in that example describe in detail three new items:

- The use of `JOptionPane.showInputDialog` to bring up a dialog box with a message and a text box for input, and returning the text typed into the box as a `String`.
- The use of Java's `Integer.parseInt` method to convert a `String` to an `int`, which is necessitated here because the `JOptionPane.showInputDialog` only returns `String` values.
- The use of Java's `JOptionPane.showMessageDialog` to bring up a dialog box to display some program output.

Nested Conditionals

There is nothing stopping us from putting conditionals inside of conditionals. Consider this decision-making problem of whether it's a good idea to cancel classes on a given day so we can go skiing.

Suppose we are only willing to go skiing if the temperature will be no higher than 50 degrees F and there is at least 6 inches of snow on the ground in the mountains.

We can ask either of the questions (temperature or snow cover) first, and if that response doesn't disqualify the day as a ski day, only then will we ask the other. Let's ask temperature first, then if the temperature is cool enough, ask about the snow cover.

All we really need to understand here is that any set of statements, including another conditional, can be placed on one of the branches of the first conditional.

See Example: ShouldWeSki

Java's `if-else-if` Construct

Our next example is a program that asks for the user's name and hometown, then displays a message that indicates whether the length of (number of characters in) the name is more than, less than, or the same as the length of the town.

We have seen most of what we need to do this, the exception being how we can compute the length of a string.

Note that there are 3 possible cases: the name is shorter, the town is shorter, or they are the same length. Since an `if` statement only has two choices, we will need more than one `if` statement.

We could accomplish this with a nested `if` as we did for the previous example. However, there is a variant on the `if-else` construct that allows us to check multiple conditions in a sequence and (optionally) perform an "otherwise" case at the end.

It is sometimes called the "if else-if" construct, and looks like this:

```
if (cond1) {
   // cond1 true stuff
}
else if (cond2) {
   // cond2 true stuff (only can happen if cond1 false)
}
else if (cond3) {
   // cond3 true stuff (only can happen if cond1 and cond2 false)
}
...
else if (condn) {
   // condn true stuff (only can happen if all previous conds false)
}
else {
   // "otherwise" -- will happen if all previous conds false
}
```

See Example: NameAndTown

In our program we can see that construct where we first check if the name is shorter. If not, we check if the name is longer. If neither was true, then they must have been equal in length, so the final `else` is executed.

Also notice that we also have a mechanism in Java to compute the length of a `String`.

Java's `String` class is very powerful, and we will see much of its functionality as we go forward. All we have done in examples so far is to declare variables capable of holding `String` references, assign `String` values to them, and use those values in constructing outputs.

If we have a `String` in a variable `s`, we can compute its length with

```
s.length();
```

Boolean data and boolean expressions

Our discussion of conditional execution needs to include a look at more complex boolean expressions.

The common boolean expression operators are expressed in Java as follows:

- arithmetic comparisons: `==` to test for equality, `!=` to test for inequality, and the inequality tests: `<`, `<=`, `>`, and `>=`.
- `&&`, which is the *and* operator. Its result is `true` if both of its operands evaluates to `true`.
- `||`, which is the *or* operator. Its result is `true` if either of its operands evaluates to `true`.
- `!` – which evaluates to the boolean opposite of its only operand.

We will encounter all of these in meaningful examples going forward, but for now, we can see many of them in action in this example.

See Example: BooleanDemo

See the comments therein to see some details.

In particular, note the precedence of these operators: `&&` is evaluated before `||`, much like multiplication is evaluated before addition in an arithmetic expression.

Important note: you need to be very careful that you do specify these operators as `&&` and `||` rather than `&` and `|`. The single-character operators will perform a bitwise and (or) rather than a logical and (or), which is not usually what you want..

Armed with these constructs and a few more we will see in this example, we can now tackle a more complicated problem.

See Example: MassPikeTolls

The comment at the top of the Java program describes the problem.

We end up with 3 possible outputs:

- There is a full toll if both entry and exit were at an interchange numbered 6 or higher, or if we are driving a truck.
- There is no toll if both entry and exit were at an interchange numbered 6 or lower, and we are not driving a truck.
- There is a toll on only part of the trip (east of interchange 6) if we entered or exited on one side of interchange 6

See the comments throughout the Java program for more information. Note in particular these new Java methods and constructs:

- The use of `System.exit(1)` to terminate the program when an error occurs (in this case, an invalid input was encountered).
- The use of a more complex form of `JOptionPage.showMessageDialog` to more clearly indicate an error message as opposed to an informational message like those we have used previously.
- The use of the `String`'s `equals` method to compare `String` values. We cannot use `==` to compare `String`s for equality in most cases. Java will accept it, but it does not have the meaning we wish it to have in this context. More on this later in the semester.

The `switch` Statement

A common pattern in programming is to have a series of statements of the form:

```
if (x == 0) {
  // do stuff for x == 0
}
else if (x == 1) {
  // do stuff for x == 1
}
else if (x == 2) {
  // do stuff for x == 2
}
...
else if (x == 8) {
  // do stuff for x == 8
}
else {
  // do stuff when x is none of the above
}
```

Let's look at an example where this occurs. Consider a program that tells you which Computer Science faculty member you can find in each of the offices in the Albertus 400 suite.

See Example: CSOfficesIfElse

Java (and many other languages) provide a special construct we can use in situations like this that can be a bit more convenient.

```
switch (x) {
  case 0:
    // do stuff for x == 0
    break;
  case 1:
    // do stuff for x == 1
    break;
  case 2:
    // do stuff for x == 2
    break;
...
  case 8:
    // do stuff for x == 8
    break;
  default:
    // do stuff when x is none of the above
    break;
}
```

Until Java version 6, this worked only when the comparison if for equality and we are using one of these data types: `char`, `byte`, `short`, or `int`. So far, we have only used `int` variables from among this group. Note that it does not work for `double`. As of Java 7, the cases can be `String` literal values.

Also note that each `case` is ended by a special statement: `break;`

If we rewrite the example to use a `switch` statement, it would look like this:

See Example: CSOfficesSwitch

If we mistakenly leave out a `break;` statement, Java will "fall through" to the next `case`. Sometimes this is handy and just what we want, but the vast majority of the time, we want a `break;` at the end of case `case`.

One situation where this does come in handy is when we want to do the same thing for multiple cases:

See Example: LittlePrimes

Formatting Output

Our next example has more conditionals, but also shows how we can nicely format output that contains floating point numbers.

See Example: Payroll

The key points to notice from this example:

- The use of named constants for numbers that are unlikely to change from one execution of the program to the next.
- The declaration of variables that will be assigned inside the `if-else` before the `if-else`. If they were defined within the body of the `if` parts and/or `else` part, those variables would exist only *within* those blocks of code.
- The declaration, construction, and use of `DecimalFormat` objects to format our floating-point output. See the text for more examples. The essentials:
  - Like `Scanner` and `JOptionPane`, we need to tell Java if we intend to use a `DecimalFormat` with

    ```
    import java.text.DecimalFormat;
    ```

  - Before we make use of one, we need to declare a variable of type `DecimalFormat` and construct an instance. The parameter we pass to this *constructor* is the number of digits and any other characters we want. There are two examples in this proram, more in the text.
  - When we want to print out a floating point value as formatted by one of these `DecimalFormat` objects, we pass the floating point value to the object's `format` method. This returns a `String` representation of that value using the specified format.

Formatting with `printf`

Another option for formatting output is the `printf` method, available in `System.out` and elsewhere. For those with C experience, it is very similar.

We used the `print` and `println` methods for console output previously. Each of those methods takes a single parameter, often a `String`, but which can be any primitive type or object type. Very often, it is passed a `String` parameter constructed by concatenating several `String` and other values.

The `printf` function (also called `format`) works a little differently. It takes a `String` called a *format string*, then an additional set of parameters that are determined by the number and order of *format specifiers* within the format string.

Section 3.11 in Gaddis has plenty of good examples and some of our later class examples will make use of `printf`.

Repetition

Another fundamental reason that computer programs are so powerful is their ability to do *repetition*.

The `while` loop

Java provides a number of *loop constructs* to support this. The `while` statement, or "while loop", is perhaps the most frequently used.

The syntax of a `while` statement is:

```
while (condition)
{
    ...
}
```

As in the `if` statement, the condition used in a `while` must be some expression that produces a `boolean` value. The statements between the open and closed curly brackets are known as the *body* of the loop.

A common way the while loop is used is as follows:

```
while (condition)
{
    do something
    change some variable so that next time you do
                    something a bit differently
 }
```

The condition controlling the `while` loop will usually involve the variable that's changing. If nothing in the condition changes, then the loop will never terminate. Such a condition is called an *infinite loop*. We avoid this, in general, by ensuring that our loops have a precise stopping condition. While we might be able to look at an algorithm and say "hey, we should stop now", Java will not (and in fact cannot, in general) determine if a loop will not stop.

Armed with this construct, we can write this program:

See Example: PerfectSquares

Other than the `while` statement itself, we see one additional Java construct here that has not come up in previous examples:

```
nextNumber++;
```

Since increment and decrement operations on variables are extremely common in programming, the designers of Java (and the designers of C before them), included a shorthand notation for these.

The above has the same effect as if we had written

```
nextNumber = nextNumber + 1;
```

There is also an example of a `printf` method call in there.

Loops for Error Checking

We will use loops in many contexts, one of which is to allow us to reissue prompts and reread input when an invalid value is entered.

To demonstrate this, we will improve on one of our old examples: the one where we determined whether a trip on the Massachusetts Turnpike was toll free, partially tolled, or fully tolled.

See Example: MassPikeTollsBetter

The changes are all at the start of the program while we input values.

See the comments there for details.

The `do-while` Loop

The `while` loop we saw in the last few examples is called a *pre-test loop*. That is, we check the condition before we enter the first time. This allows a `while` loop to execute its body 0 times if the condition is initially false.

In some circumstances, we want to execute the loop at least once. Such a loop is called a *post-test loop*.

Consider the problem where we have a sequence of numbers to read in, say prices of items at a supermarket checkout, for which we want to keep a running total to report at the end.

Java provides a construct we can use for this purpose – the `do-while` loop.

It is basically the same as a `while` loop, except we begin it with the keyword `do`, follow with the body of the loop, and end it with a `while` keyword and condition.

```
do
{
    ...
} while (condition);
```

See Example: Checkout

This example demonstrates the `do-while` construct.

One other Java construct here that we have not yet used is the `+=` assignment operator:

```
total += itemPrice;
```

Much like the `++` we saw recently for the increment operation (and the corresponding `--` operation for decrement), this is a shorthand notation for a common programming task: adding a value to a variable and storing the result back in that variable:

```
total = total + itemPrice;
```

This shorthand exists for all of the standard arithmetic operators: `-=`, `*=`, `/=` and `%=`.

For example, if we wanted to double the value in a variable `x`, we could use the shorthand:

```
x *= 2;
```

You are never going to be required to use these shorthand operators, but they are convenient, and you will need to recognize them in my examples.

Counting Loops

All of the loops we wish to have in our programs can be written using the `while` and `do-while` constructs we have just seen.

However, most programming languages include another construct that is typically used for *counting loops*. Java has such a construct, called a *for loop*.

Java's `for` loop looks a bit different, but essentially has all of the same components.

```
for (int number = 1; number <= 10; number++)
{
   // do stuff - but omit number++ at end
}
```

The code in the parentheses consists of 3 parts; it is not just a condition as in `if` or `while` statements. The parts are separated by semicolons. The first part is executed once when we first reach the `for` loop. It is used to declare and initialize the counter. The second part is a condition, just as in `while` statements. It is evaluated before we enter the loop (ı,e it is a pre-test loop) and before each subsequent iteration of the loop. It defines the stopping condition for the loop, comparing the counter to the upper limit. The third part performs an update. It is executed at the *end* of each iteration of the `for` loop, just before testing the condition again. It is used to update the counter.

We use the loop above in a straightforward example: calculating the sum of the first 10 integers.

See Example: Sum1To10

Notice how the `for` localizes the use of the counter. This has two benefits. First, it simplifies the body of the loop so that it is somewhat easier to understand the body. More importantly, it becomes evident, in one line of code, that this is a counting loop.

Other variations

Many variations are possible and we will see them frequently throughout the remainder of the course. For example, we could *count down* instead of up:

See Example: Countdown

This includes not only a count down loop, but a loop whose starting condition depends on the value in a variable instead of an integer constant. We can use any arithmetic expression for the initialization and any boolean expression for the stopping condition.

If we wanted to count by 2's to add up the even numbers:

See Example: Sum2ToNBy2

We can compute some number of terms of the geometric sum

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + ...$$

If we continue this sum infinitely, the series sums to 1 (can you prove it?).

See Example: GeometricFractionalSum

This example has a straightforward counting loop structure, but has more work to do each time around the loop. Not only do we need to make sure we iterate the proper number of times, we also need to update the value of the next term to be added each time around.

Random Numbers

It is often useful have computer programs choose *random numbers*. Programs that implement games might need to make decisions randomly. This could involve choosing a random direction for a character to move, an order for shuffling a deck of cards, or to simulate the roll of a die.

We will see how this works in Java by looking at an example:

See Example: RandomDemo

If we want to use random numbers in our program, we need to construct a `Random` object that we can ask to generate our random numbers. This first requires that we add an appropriate `import` statement:

```
import java.util.Random;
```

Then in our `main` method, we construct an instance:

```
  Random randomGenerator = new Random();
```

We can then get random values from our `Random` object by calling its methods including `nextInt` and `nextDouble`. See the RandomDemo example code for specifics.

A Random Number in a Game

We can use this capability in many ways. We will first implement a simple guessing game. We will have the computer pick a random number between 1 and 100, and the user gets to make repeated guesses until the guess is correct. The program helps out by giving a "higher" or "lower" response.

See Example: GuessingGame

Here, we just need to choose our random number for the answer, then have a loop that reads guesses until the correct number is entered.

A Monte Carlo Method to Compute π

Not only games make use of random numbers. There is a class of algorithms knows as *Monte Carlo methods* that use random numbers to help compute some result.

We will write programs that use a Monte Carlo method to estimate the value of π.

The algorithm is fairly straightforward. We repeatedly choose (x, y) coordinate pairs, where the x and y values are in the range 0-1 (*i.e.*the square with corners at $(0, 0)$ and $(1, 1)$. For each pair, we determine if its distance from $(0, 0)$ is less than or equal to 1. If it is, it means that point lies within the first quardant of a unit circle. Otherwise, it lies outside. If we have a truly random sample of points, there should be an equal probability that they have been chosen at any location in our square domain. The space within the circle occupies $\frac{\pi}{4}$ of the square of area 1.

So we can approximate π by taking the number of random points found to be within the unit circle, dividing that by the total number of points and multiplying it by 4!

We can see a simulation of this at:

On the web: http://en.wikipedia.org/wiki/File:Pi_30K.gif at

Our program:

See Example: MonteCarloPi

Note that we can simply call our `Random`'s `nextDouble` method to get numbers in the 0.0 to 1.0.