CSSE 220 Session 01

Student Intros
Brief Course Intro
Java Intro

Agenda

- Roll Call
- A few administrative details
- Tour of Course materials
- Homework 01 overview
- Java vs. Python and C
- A first Java program (PrintHello)
- Factorial program(s)

Daily Quizzes

- Why do we have them?
- I expect you can answer every question.
- Stop me if I don’t cover a question!

Roll Call/Introductions

- If I mispronounce your name, or if you want to be called by another name than what the Registrar gave me, please tell me.
- While I am calling the roll, register your attendance on ANGEL.
  - I'll write the PIN on the board.
  - Do this every class day.
- Tell us something about yourself:
  - hometown
  - something that most people in the room probably do not know about you
- Instructor Introduction: Session 2

Q1, 2

Contact Me

- Office (F-210)- I am there a lot, when I am not in meetings.
  - My Outlook Calendar is public, and a web version is also linked from the Syllabus (and from HW1).
  - Sometimes I'll be found in F217 helping students.
- Phone – x8331.
- Email: anderson@rose-hulman.edu
  - See syllabus for instructions about subject lines
- Better: csse220-staff@rose-hulman.edu will go to me and the student assistants.
- ANGEL discussion forums
- I want to help you (really!)
  - and I also hired a small army of students to help.

Q3-6

Student Assistant Lab Hours

- Place and Times
  - F-217
  - Sunday-Thursday 7-9 PM (at least)
  - MTWRF afternoons
  - http://www.rose-hulman.edu/class/csse/resources/Labs/coverage.htm
  - Should begin today

Q7

Big Java Textbook

- Excellent introduction to Object-oriented programming in Java.
- Good mix of theory and practice, design and implementation.
- Some challenging problems, a good place to go as you review for exams.
- Read it!

- It is a textbook written for **beginners**.
  - How should you read the first few chapters?

HW 1: Due session 2 (due 8:05 AM day of Session 2)

- Read course Syllabus.
- Preview the **Python** *vs* C document.
- Read the wiki instructions.
- Reading from Chapter 1 and the first part of Chapter 2 of *Big Java.*
- Answer questions for the wiki, as described in the wiki instructions.
- Optional course setup items.
- Install software if necessary.
- Complete and run programs from this session and add another simple program.

Programming is not a spectator sport

And neither is this course.

Ask, evaluate, respond, comment!

Is it better to ask a question and risk revealing your ignorance, or to remain silent and perpetuate your ignorance?

Feel free to interrupt during class discussions

- Even with statements like, “I have no idea what you were just talking about.”
- We want to be polite, but in this room learning trumps politeness.
- I do not intend for classroom discussions to go over your head. Don't let them!

Some major programming ideas that you should already know*

- variables, types, type conversions, assignment
- if-**else** statements, **while** and **for** loops, **break**
- functions
  - formal and actual arguments
  - pass-by-value
  - returning a value *vs.* doing something
  - local variables
- Arrays and lists
- Basics of classes and objects
  - instance variables and class variables
  - methods, encapsulation, constructor
  - **self** (in Java, it's called **this**)

*and mostly remember. The textbook will review them all, and we'll quickly review most of them in class

More Administrivia Tomorrow

- After you have had a chance to read the syllabus.
  - If you have questions on it, write them down so you'll remember to ask about them in class.
  - Same thing for reading from the textbook.
- Also an introduction to me.

A Tour of the Online Course Materials

Q8-10

Break

- We will take a 3-5 minute break each day.
- I strongly suggest that you at least stand up.
- Better: walk around for a minute or so.

CSSE220 is not primarily about Java

- The emphasis is object-oriented development, with Java as a medium.
- Covering everything about Java is **not** a goal of this course.
- We will focus mainly on features that are common to object-oriented languages.

Things Java has in common with Python

- Classes and objects.
- Lists (but no special language syntax for them like in Python).
- Standard ways of doing graphics, GUIs.
- A huge library of classes/methods that make many tasks easier.
- A better Eclipse interface than C has.
    - Eclipse was "made for Java".

Things Java has in common with C

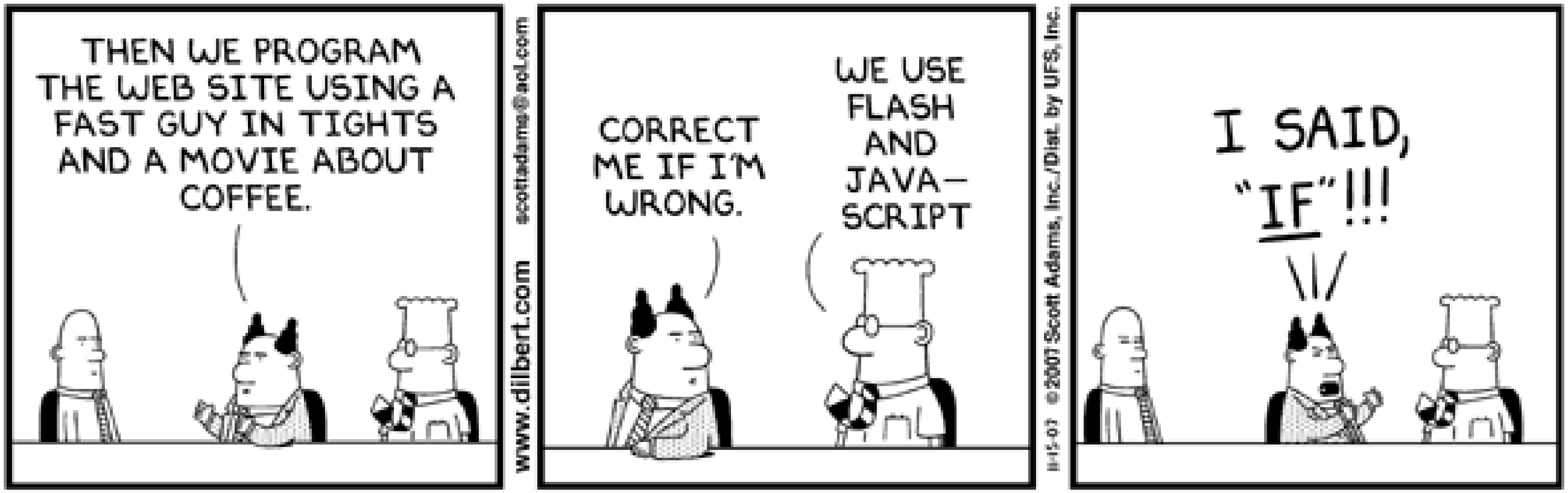
- Many similar primitive types:
  - int, char, long, float, double, ….
- Static typing.
  - Types of all variables must be declared.
- Similar syntax and semantics for **if**, **for**, **while**, **break**, **continue**, and function definitions.
- Semicolons mostly required in the same places.
- Execution begins with the **`main()`** function.
- Comments: **`//`** and **`/* … */`**
- Arrays are homogeneous, and their (fixed) sizes must be declared when we create them.

Why Java?

- Widely used in industry for large projects
  - From cell phones
  - To global medical records
- Object-oriented (unlike C)
- "Statically type safe" (unlike Python, C, C++)
- Less complex than C++
- Part of a strong foundation

Q11

Interlude


THEN WE PROGRAM THE WEB SITE USING A FAST GUY IN TIGHTS AND A MOVIE ABOUT COFFEE.
scottadams@aol.com
www.dilbert.com
CORRECT ME IF I'M WRONG.
WE USE FLASH AND JAVA-SCRIPT
© 2007 Scott Adams, Inc./Dist. by UFS, Inc.
I SAID, "IF"!!!

A First Java Program

In Java, all variable and function definitions must be inside class definitions

main is where we start

```
public class HelloPrinter {

    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }

}
```

System.out is Java's standard output stream. Note that this is the variable called **out** in the **System** class

System.out is an object from the **PrintStream** class. **PrintStream** has a method called **println( )**

Q12

Let's Get Started!

- You should already have on your laptop:
  - **Java 6** (a.k.a JDK 1.6)
  - **Eclipse 3.4** (make sure you have this version!)
  - **Subclipse**
  - If not, look on with someone else today.
  - Installation instructions are in HW.
- Then go to Homework 1, part 7a,b
  - Create an Eclipse Workspace in which to put your programs for this course.
  - Go to your SVN repository in Eclipse SVN Repository Browsing Perspeective.
  - Check out today's project.
  - Switch to Java Perspective.
- Try to figure out how to run **HelloPrinter.java**
- Get help if you're stuck!

HelloPrinter.java

- To run a Java program:
  - Right-click it in the Package Explorer view
  - Choose **Run As → Java Application**
- Change the program to say hello to a person next to you
- Introduce an error in the program
  - See if you can come up with a different error than the person next to you
- Fix the error that the person next to you introduced

A Second Java Program

Define a constant, MAX

Except for **public static**, everything about this function definition is identical to C.

We can declare the loop counter variable in a **for** loop's header.

println terminates the output line after printing; print does not.

```
// Author:  Claude Anderson.  Nov 19, 2007.

public class Factorial {

   public static final int MAX = 17;

   /* Returns the factorial of n */
   public static int factorial (int n) {
      int product = 1;
      int i;
      for (i=2; i<=n; i++) {
         product = product * i;
      }
      return product;
   }

   public static void main(String[] args) {
      for (int i=0; i <= MAX; i++) {
         System.out.print(i);
         System.out.print("! = ");
         System.out.println(factorial(i));
      }
   }
}
```

Q13-14

Enter and Run Factorial.java

- Get help if you get stuck!
- What happens when **i** gets to 14?

Q15

Additional slides for later reference

Q16-18

The next eight slides

- Illustrate things we will code "live" in class
- In the unlikely chance that we get done everything else before the class period ends.
- They are here for you to refer to later.

A (slightly different)factorial program

In Java, all variable and function definitions are inside class definitions.

Define a constant, MAX

Except for **public static**, everything about this function definition is identical to C.

Note the function signature for Java's **main()** .

We can declare the loop counter in **for** loop header.

println terminates the output line after printing; **print** does not.

System.out is Java's standard output stream. Note that this is the variable called **out** in the **System** class.

```
// Author:  Claude Anderson.  Nov 19, 2007.

public class Factorial {

    public static final int MAX = 17;

    /* Returns the factorial of n */
    public static int factorial (int n) {
        int product = 1;
        int i;
        for (i=2; i<=n; i++) {
            product = product * i;
        }
        return product;
    }

    public static void main(String[] args) {
        for (int i=0; i <= MAX; i++) {
            System.out.print(i);
            System.out.print("! = ");
            System.out.println(factorial(i));
        }
    }
}
```

System.out is an object from the **PrintStream** class. **PrintStream** has methods called **print( )** and **println( )** .

A runof the Factorial Program

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 1932053504
14! = 1278945280
15! = 2004310016
16! = 2004189184
17! = -288522240
```

What happens when i gets to 14?

```java
// Author:  Claude Anderson.  Nov 19, 2007.

public class Factorial {

   public static final int MAX = 17;

   /* Returns the factorial of n */
   public static int factorial (int n) {
      int product = 1;
      int i;
      for (i=2; i<=n; i++) {
         product = product * i;
      }
      return product;
   }

   public static void main(String[] args) {
      for (int i=0; i <= MAX; i++) {
         System.out.print(i);
         System.out.print("! = ");
         System.out.println(factorial(i));
      }
   }
}
```

Larger Factorials: the long type

It still overflows, but not as quickly.

```
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = -4249290049419214848
```

```java
public class Factorial_2_WithLongs {
  public static final int MAX = 21;

  /* Return the factorial of n */
  public static long factorial (int n) {
    long product = 1;
    for (int i=2; i<=n; i++)
      product *= i;
    return product;
  }

  public static void main(String[] args) {
    for (int i=0; i <= MAX; i++)
      System.out.println(i + "! = " + factorial(i));
   }
}
```

static: Not associated with any particular object.

A Java **int** is a 32-bit signed integer; a **long** is a 64-bit signed integer.

If either operand is a String, + is the concatenation operator.

If the other argument of + is not a string, that argument is automatically converted to a String (unlike in Python, where you must explicitly call `str()` to do the conversion).

Huge Factorials With BigInteger

Java's **BigInteger** is like Python's **long** type. There is no set limit on how large a BigInteger can be.
But calculations are less efficient than with Java's **int** or **long** types.

The **BigInteger** class is imported from the **java.math** package.

```
import java.math.BigInteger;

public class Factorial_3_BigInteger {

  public static final int MAX = 100;

  /* Return the factorial of n */
  public static BigInteger factorial(int n) {
    BigInteger prod = BigInteger.ONE;
    for (int i=2; i<=n; i++)
       prod = prod.multiply(new BigInteger(i + ""));
    return prod;
  }

  public static void main(String[] args) {
     for (int i=0; i <= MAX; i++)
        System.out.println(i + "! = " + factorial(i));
   }
}
```

ONE is the name of a **BigInteger** constant (that represents the integer 1).

new BigInteger(someString) calls the BigInteger constructor that takes a String argument.

multiply() is a method of the BigInteger class that takes a BigInteger object as its argument, and returns the product as a new BigInteger object.

i+ "" is a quick and easy way to get from a number to its **String** representation.

final means that the value of this variable can never change. So it is treated as a constant.

the BigInteger object returned by **factorial()** can be automatically convertet to a String because BigInteger has a **toString()** method.

Output in Columns: Fixed Width

```
 0                               1
 1                               1
 2                               2
 3                               6
 4                              24
 5                             120
 6                             720
 7                            5040
 8                           40320
 9                          362880
10                         3628800
11                        39916800
12                       479001600
13                      6227020800
14                     87178291200
15                   1307674368000
16                  20922789888000
17                 355687428096000
18                6402373705728000
19              121645100408832000
20             2432902008176640000
21            51090942171709440000
22          1124000727777607680000
23         25852016738884976640000
24        620448401733239439360000
25     15511210043330985984000000
```

```java
import java.math.BigInteger;

public class Factorial_4_Printf {

   public static final int MAX = 25;

   /* Return the factorial of n */
   public static BigInteger factorial(int n) {
      BigInteger prod = BigInteger.ONE;
      for (int i=1; i<=n; i++)
        prod = prod.multiply(
                 new BigInteger(i + ""));
      return prod;
    }

   public static void main(String[] args) {
      for (int i=0; i <= MAX; i++)
         System.out.printf("%2d  %30s\n", i,
                           factorial(i));
   }
}
```

The syntax and semantics of **printf** in Java and C are identical for simple output formats. The format strings in Java and Python are also the same

Output in Columns: Calculated Width

```
 0                           1
 1                           1
 2                           2
 3                           6
 4                          24
 5                         120
 6                         720
 7                        5040
 8                       40320
 9                      362880
10                     3628800
11                    39916800
12                   479001600
13                  6227020800
14                 87178291200
15               1307674368000
16              20922789888000
17             355687428096000
18            6402373705728000
19          121645100408832000
20         2432902008176640000
21        51090942171709440000
22      1124000727777607680000
23     25852016738884976640000
24    620448401733239439360000
25  15511210043330985984000000
```

```java
import java.math.BigInteger;

public class Factorial_5_CalculateWidth {
  public static final int MAX = 25;

  public static BigInteger factorial(int n) {
    BigInteger prod = BigInteger.ONE;
    for (int i=1; i<=n; i++)
      prod = prod.multiply(new BigInteger(i +""));
      return prod;
    }

  public static void main(String[] args) {
    int len = factorial(MAX).toString().length();

    for (int i=0; i <= MAX; i++)
      System.out.printf( "%2d  %" + len + "s\n",
                         i,
                         factorial(i));

   }
}
```

Ask user for value (new way)

Import the **Scanner** class from the **java.util** package.

System.in is Java's standard input stream. Note that this means the variable called **in** in the **System** class.

Other **Scanner** methods include nextDouble(), nextLine(), nextBoolean, hasNextInt(). hasNextline().

```
import java.math.BigInteger;
import java.util.Scanner;

public class Factorial_6_Scanner {
   public static final int MAX = 25;

   public static BigInteger factorial(int n) {
     BigInteger prod = BigInteger.ONE;
     for (int i=1; i<=n; i++)
       prod = prod.multiply(new BigInteger(i +""));
     return prod;
   }

  public static void main(String[] args) {
     Scanner sc = new Scanner(System.in);
     System.out.print("Enter a nonnegative integer: ");
     int n = sc.nextInt();
     System.out.println(n + "! = " + factorial(n) );
  }
}
```

If we do not do the import, we can write

```
   java.util.Scanner sc = new java.util.Scanner(System.in);
```

So **import** is a simple convenient shortcut

Ask user for value (old way)

Using the new Scanner class is easier than this approach. But you will often see the old approach in other people's code (including Mark Weiss' code).

Think of this as the "magic incantation" for getting set up to read from standard input.

readline( ) returns the next line of input as a String.

Since **readline( )** could generate an IO Exception, the try/catch is required.

```java
import java.math.BigInteger;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.BufferedReader;

  // omitted definition of the factorial method

  public static void main(String[] args) {
     BufferedReader in =
        new BufferedReader(
         new InputStreamReader(System.in));
     String line = "";
     System.out.print("Enter a positive integer: ");
     try {
        line = in.readLine();
     } catch (IOException e) {
        System.out.println("Could not read input");
     }
     int n = Integer.parseInt(line);
     System.out.println(n + "! = " + factorial(n) );
  }
}
```

parseInt( ) takes a string that represents an integer and returns the corresponding int value. It is somewhat similar to Python's **int( )** function.

The next five slides

- Are not things you'll need to know for this course.
- But are here for completeness.
- Mainly for those who like to compile run programs from the command line…
- … or who want to use a Java program as part of a script.

Command-line arguments

- Sometimes a program is not run standalone or from within an Integrated Development Environment.
- Sometimes it is run as part of a script, fromr a command-line, such as a UNIX/Linux shell or a Windows DOS Prompt.
- The **args** parameter of the Java **main()** method allows us to access the command-line arguments.
- **args[0]** is the first command-line argument, **args[1]** the second, etc.

Add Java to your Windows Path

- You need to have the folder containing **javac.exe** in your Path* variable:
- If you have not already added it:
  - Right-click **My Computer**, choose **Properties**
  - **Advanced**, then **Environment Variables**
  - Under **System variables**, click **Path**, then click **Edit**
  - Click in **Variable value** field, then press the End key
  - Add a **semicolon**, followed by the path to your Java **bin** folder, which should be something like
    C:\Program Files\Java\jdk1.6.0_01\bin
  - Click **OK** several times to exit.

*Path is a list of folders in which the system looks for programs to run.

Compile and Run a Java Program From the Command line

- First, **cd** to the folder where the file lives.
- **javac ClassName.java** compiles the file and produces **ClassName.class** .
- **java ClassName** executes that class.
- **java ClassName arg0 arg1** … executes that class with the given command-line arguments.

```
C:\Documents and Settings\anderson\Desktop> javac Factorial_8_CommandLine.java
C:\Documents and Settings\anderson\Desktop> java Factorial_8_CommandLine 8
8!=40320
C:\Documents and Settings\anderson\Desktop> java Factorial_8_CommandLine 120
120!=66895029134491270575881180540903725867527463331380298102956713523016335572449629893668741652719849813081576378932140905525344085894081218598984811143896500059649605212569600000000000000000000000000000
```

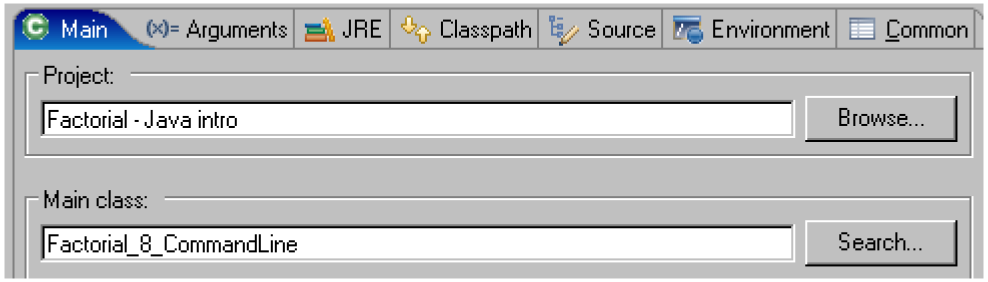
Factorial with command-line argument

```
import java.math.BigInteger;

public class Factorial_8_CommandLine {
   public static final int MAX = 25;

   public static BigInteger factorial(int n) {
     BigInteger prod = BigInteger.ONE;
     for (int i=1; i<=n; i++)
       prod = prod.multiply(new BigInteger(i +""));
     return prod;
   }

   public static void main(String[] args) {
     int n = Integer.parseInt(args[0]);
     System.out.println(n + "! = " + factorial(n));
   }
}
```

args[0] is the first command-line argument.

Command-line arguments in Eclipse

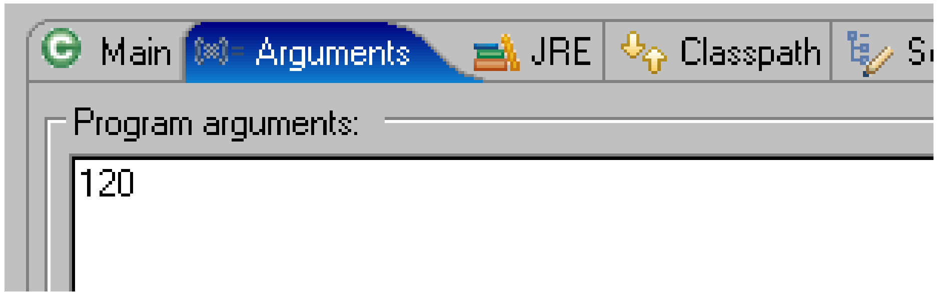
- **Run as … Run …**
- On Main tab, make sure the class you want to run is selected. If not, **use Search …**



Project:
Factorial - Java intro

Main class:
Factorial_8_CommandLine

- Click **Arguments** tab.
- Enter the Arguments under **Program Arguments.**
- Click Run.



Program arguments:
120

The remaining slides

- Are previews of tings we'll discuss soon
- View them if you are interested.

What if a user types something wrong?

If any exception gets thrown by the code in the **try** clause, the **catch** clauses are tested in order to find the first one that matches the actual exception type.

If none match, the exception is thrown back to whatever method called this one.

If it is never caught, the program crashes.

```
import java.math.BigInteger;

public class Factorial_9_InputErrors {

   public static BigInteger factorial(int n) {
      if (n < 0)
         throw new IllegalArgumentException();
      BigInteger prod = BigInteger.ONE;
      for (int i = 1; i <= n; i++)
         prod = prod.multiply(new BigInteger(i + ""));
      return prod;
   }

   public static void main(String[] args) {
      try {
         int n = Integer.parseInt(args[0]);
         System.out.println(n + "! = " + factorial(n));
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Command-line arg required");
      } catch (NumberFormatException e) {
         System.out.println("Argument must be an integer");
      } catch (IllegalArgumentException e) {
         System.out.println("Argumentcannot be negative");
      }
   }
}
```

Factorial recursive

Recursive factorial definition:
n! = 1 if n = 0
n! = (n-1)! n if n>0

```
import java.math.BigInteger;

public class Factorial_10_Recursive {
   public static final int MAX = 30;

   /* Return the factorial of n */
   public static BigInteger factorial(int n) {
      if (n < 0)
         throw new IllegalArgumentException();
      if (n == 0)
         return BigInteger.ONE;
      return new BigInteger(n+ "").multiply(factorial(n-1));
    }

   public static void main(String[] args) {
      for (int i=0; i <= MAX; i++)
         System.out.println(i + "! = " + factorial(i) );
   }
}
```

Recursive basically means:
The method calls itself.

Speed up Factorial Calculation with Caching

Store previously-computed values in an array called vals

```
import java.math.BigInteger;

public class Factorial_11_Caching {
   public static final int MAX = 30;
   static int count = 0;    // How many values have we cached so far?
   static BigInteger [] vals = new BigInteger[MAX+1];  // the cache
   static { vals[0] = BigInteger.ONE; }          // Static initializer

   /* Return the factorial of n */
   public static BigInteger factorial(int n) {
      if (n < 0 || n > MAX)
         throw new IllegalArgumentException();
      if (n <= count)  // If we have already computed it …
         return vals[n];
      BigInteger val =
         new BigInteger(n+ "").multiply(factorial(n-1));
      vals[n] = val;  // Cache the computed value before returning it
      return val;
  }
               // Code for main()omitted. Same as in previous example.
}
```

File Input/Output

```
import java.util.*;
import java.io.*;

public class FileIOTest {
/* Copy an input file to an output file, changing all letters to uppercase.
   This approach can be used for input processing in almost any program. */
   public static void main(String[] args) {
      String inputFileName =  "sampleFile.txt";
      String outputFileName = "upperCasedFile.txt";
      try {
         Scanner sc = new Scanner(new File(inputFileName));
         PrintWriter out = new PrintWriter(new FileWriter(outputFileName));
         while (sc.hasNextLine()){  // process one line
            String line = sc.nextLine();
            line = line.toUpperCase();
            for (int i= 0; i< line.length(); i++)
            // normally we might do something with each character in the line.
               out.print(line.charAt(i));
            out.println();
         }
         out.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

More File Input/Output

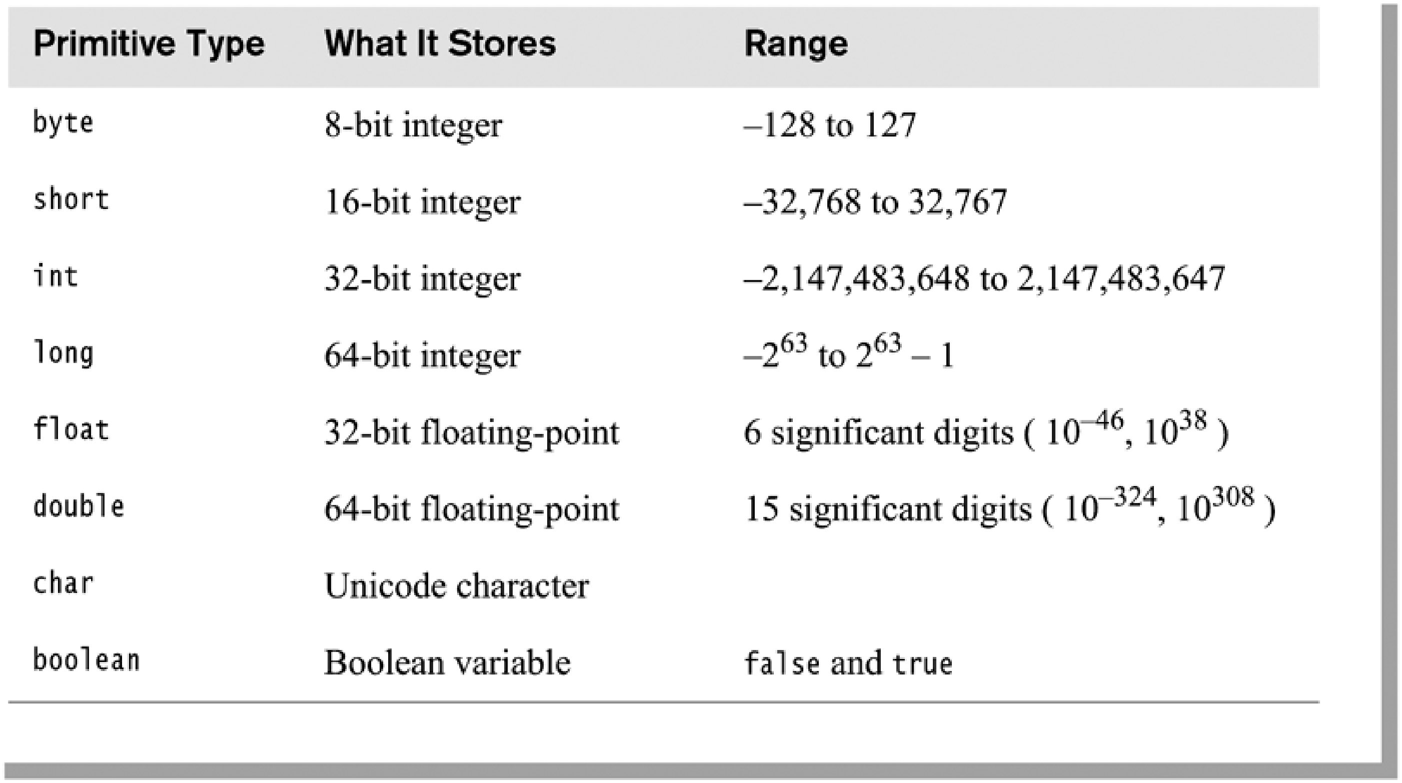
```
import java.util.Scanner;
import java.io.*;

public class TryFileInputOutput {

  public static void main(String[] args) {
    String inFileName=null ,outFileName = "outFile.txt";
    Scanner fileScanner;
    PrintWriter out;
    try {
      Scanner sc = new Scanner(System.in);
      while (true) // until we get a valid file.
        try {
          System.out.print("Enter input file name: ");
          inFileName = sc.nextLine();
          fileScanner = new Scanner(new File(inFileName));
          break; // we have a valid file, so exit the loop.
        } catch(FileNotFoundException e) {
          System.out.println("Did not find file " + inFileName + ". Try again!");
      }
      out = new PrintWriter(new FileWriter(outFileName));
      while (fileScanner.hasNextLine()){  // process one line
        String line = fileScanner.nextLine();
        line = line.toUpperCase();
        for (int i=0; i<line.length(); i++)
          out.print(line.charAt(i));  // process each char on the line
        out.println();
      }
      out.close();
      fileScanner.close();
      System.out.println("Done!");
    } catch (IOException e) {
      e.printStackTrace();
    }
```

Essentially the same as before

Keep looping until user enters the name of an input file that we can actually open.

Essentially the same as before

Primitive types



Primitive Type	What It Stores	Range
`byte`	8-bit integer	–128 to 127
`short`	16-bit integer	–32,768 to 32,767
`int`	32-bit integer	–2,147,483,648 to 2,147,483,647
`long`	64-bit integer	-2^{63} to $2^{63}-1$
`float`	32-bit floating-point	6 significant digits (10^{-46}, 10^{38})
`double`	64-bit floating-point	15 significant digits (10^{-324}, 10^{308})
`char`	Unicode character	
`boolean`	Boolean variable	`false` and `true`

figure 1.2

The eight primitive types in Java

Java *switch* statement

figure 1.5

Layout of a switch statement

```
switch( someCharacter )
{
  case '(':
  case '[':
  case '{':
    // Code to process opening symbols
    break;

  case ')':
  case ']':
  case '}':
    // Code to process closing symbols
    break;

  case '\n':
    // Code to handle newline character
    break;

  default:
    // Code to handle other cases
    break;
}
```

Ternary conditional operator ? :

```
public class MinTest
{
    public static void main( String [ ] args )
    {
        int a = 3;
        int b = 7;

        System.out.println( min( a, b ) );
    }

    // Method declaration
    public static int min( int x, int y )
    {
        return x < y ? x : y;
    }
}
```

figure 1.6

Illustration of method declaration and calls