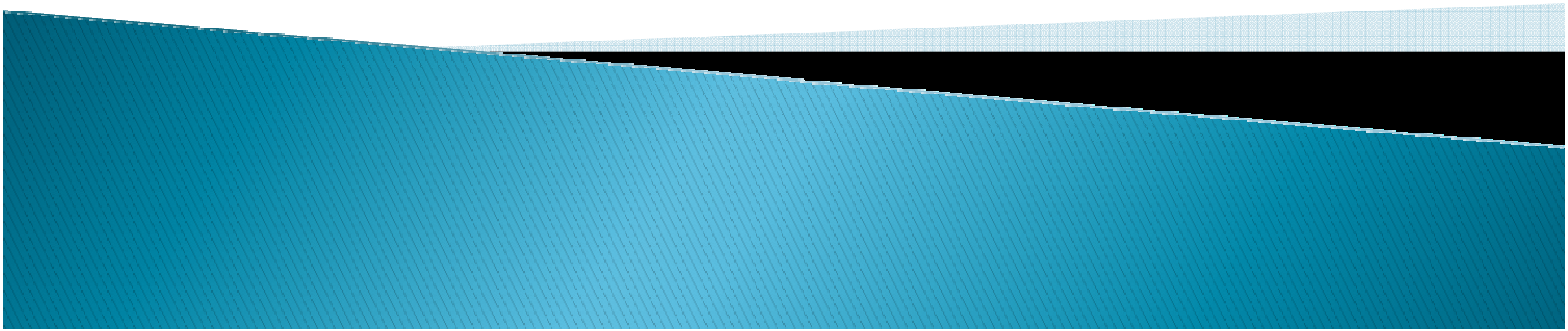
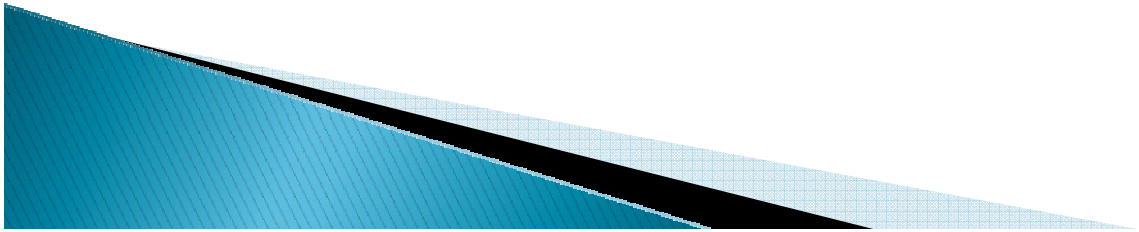
CSSE 220 Day 1

Brief Course Intro
Instructor Intro
Java Intro



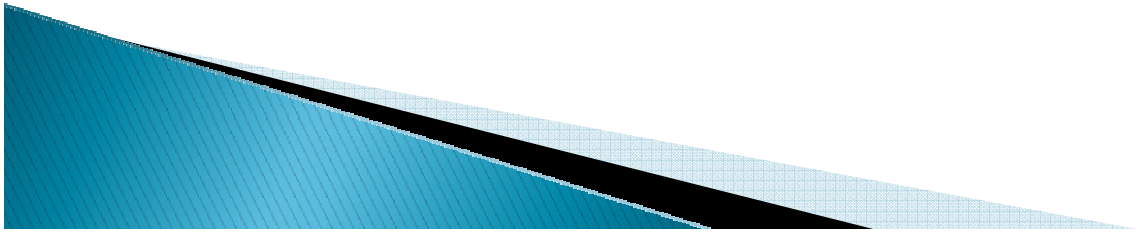
Agenda

- Roll Call
- A few administrative details(more next time)
- Java vs. Python and C
- A first Java program (calculate factorials)
- Some factorial variations
- A new operator (? :)
- A look at the homework



Daily Quizzes

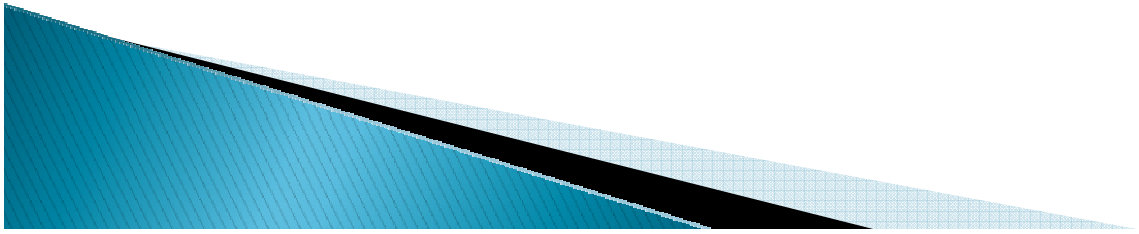
- We will have them most days.
- Help you interact with the lecture material.
- Answers should be in the PowerPoint slides and class discussion.
- When I return them, they should be notes for you.
- A way for you to give me feedback, ask questions, or let me discover that we need more class time on a topic.



Roll Call

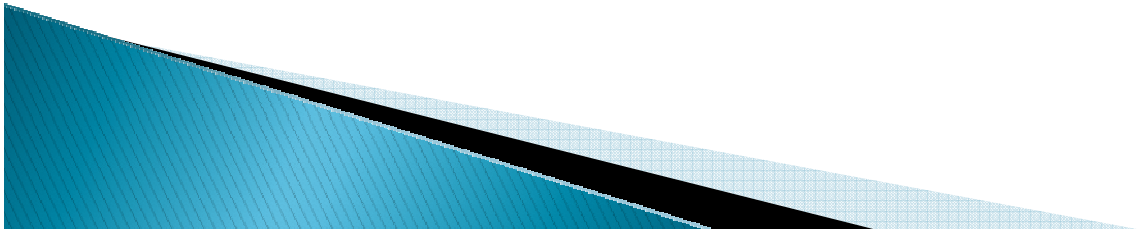
- Me: Matt Boutell

- If I mispronounce your name, or if you want to be called by another name than what the Registrar gave me, please tell me.



A quick tour of the online course materials

- This is only the second time this new version of the course has been taught!
  - Borrowed heavily from Claude Anderson's materials from Winter term
- I will usually post my PowerPoint slides after each class meeting.
  - If I ever forget, feel free to remind me.
- Let's go to Angel

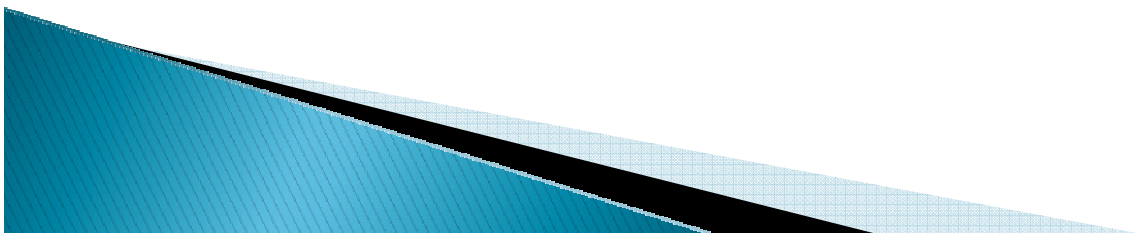


Programming is not a spectator sport

And neither is this course.

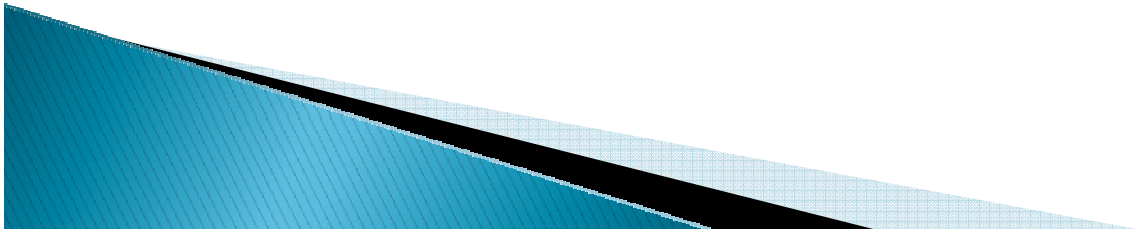
Ask, evaluate, respond, comment!

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



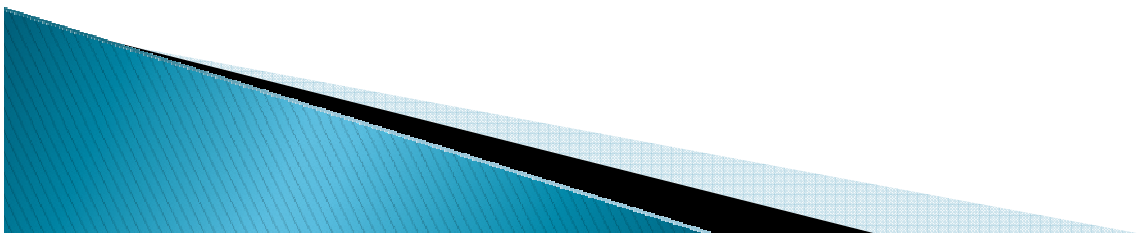
Feel free to interrupt during class discussions

- 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!



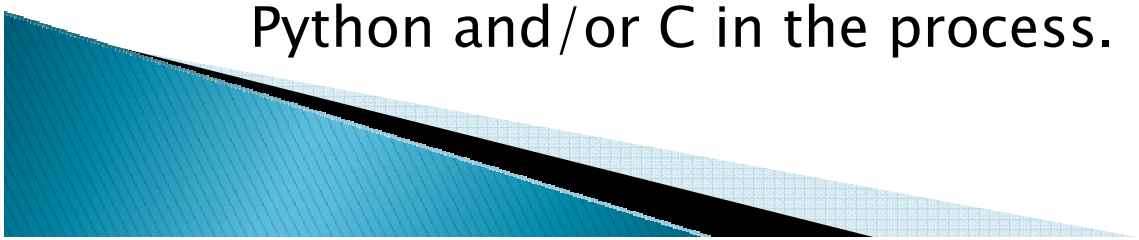
More Administrivia Tomorrow

- That's because we want time to program now!
  - Remember, bring questions to class.
  - Same thing for reading from the textbook.
- Any other pressing questions before we dive in?



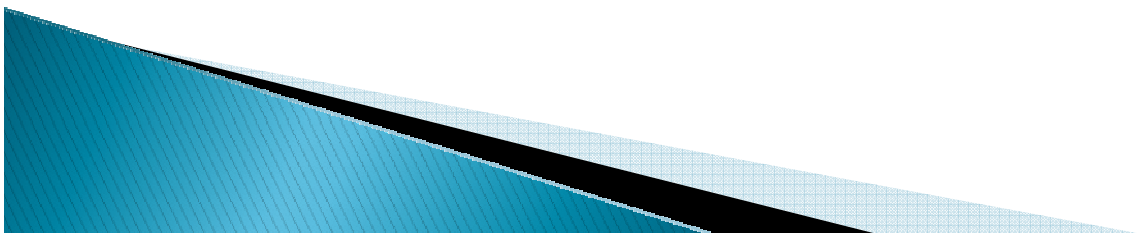
Two Audiences for the Java Intro

- Some of you know some Java from taking CSSE 120 previous to Fall, 2007.
  - Most of the Java intro will be review.
  - But don't go to sleep:
    - a few things are likely to be new,
    - or be rusty in your mind because it has been a while since you did Java programming.
- Most of you know some Python and C.
  - I assume that Java is unknown to you.
  - We can move fast because of what you do know.
  - I'll sometimes compare/contrast Java with Python or C.
  - Folks from the other group should not need that analogy, but if you wish you can learn a little about Python and/or C in the process.



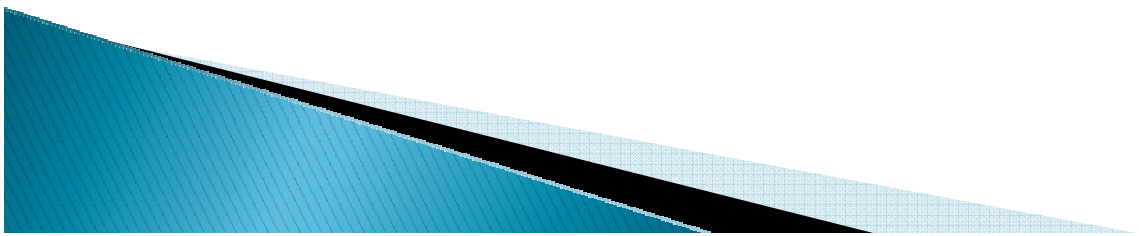
Things Java Has in Common with Python

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



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**, function definitions.
- Semicolons required mostly in the same places.
- Execution begins with the main() function.
- Comments: `//` and `/* … */`
- Arrays are homogeneous, and size must be declared at creation.

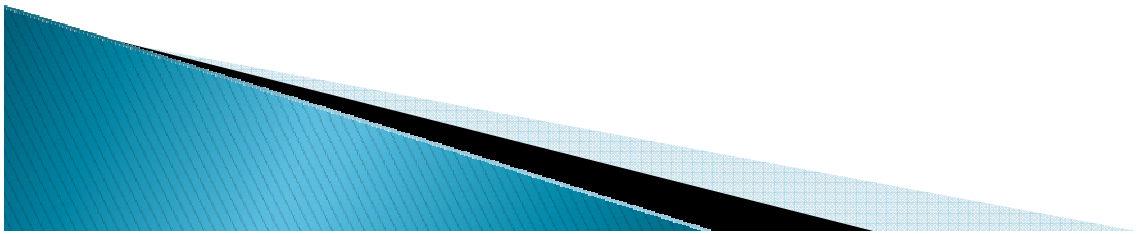


Configuring Eclipse for Java

- See

http://www.rose-hulman.edu/class/csse/resources/Eclipse/eclipse-java-configuration.htm

  - Create a new workspace
  - Download preferences
  - Point to local javadoc



A First Java 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.

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

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

public class Factorial_1_FirstJavaProgram {

   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));
      }
   }
}
```

Run the First Java 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?

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

public class Factorial_1_FirstJavaProgram {

   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
```

```
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.

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

```
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));
  }
}
```

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
```

```
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));
   }
}
```

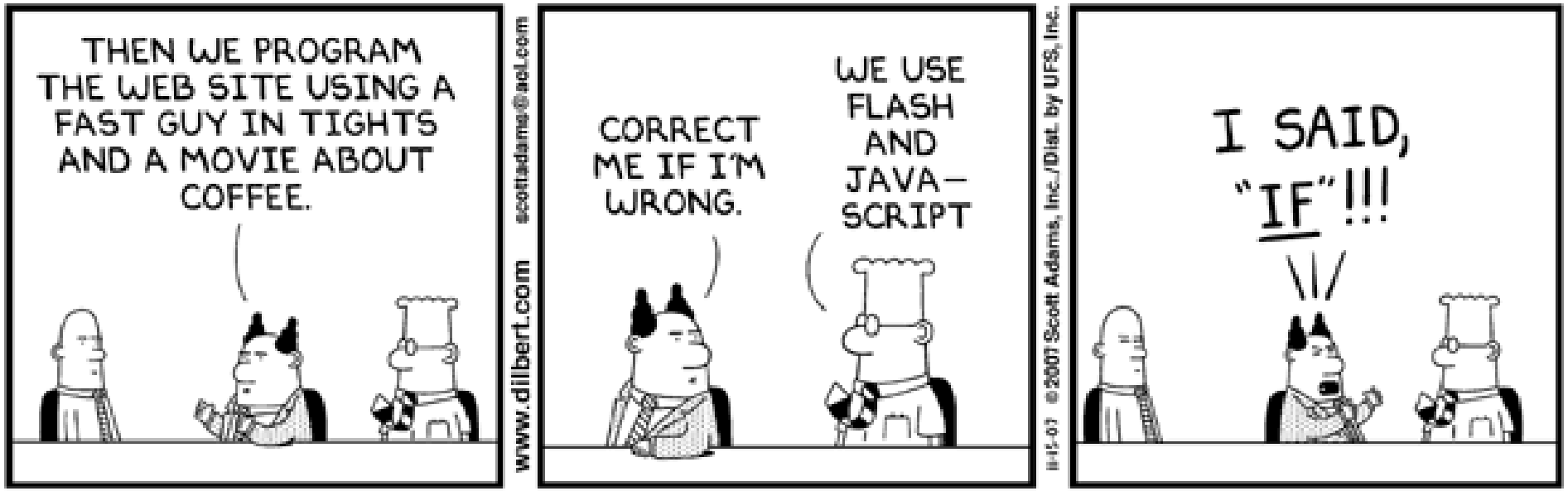
The syntax and semantics of **printf** in Java and C are identical for simple output formats. The format strings in Java and Python are also the same

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
```

```
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));
   }
}
```

Interlude



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.

```
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.

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) {
      Scanner scanner = new Scanner(System.in);
      System.out.print("Enter a nonnegative integer: ");
      try {
        int n = scanner.nextInt();
        System.out.println(n + "! = " + factorial(n));
      } catch (InputMismatchException e) {
         System.out.println("Argument must be an integer");
      } catch (IllegalArgumentException e) {
         System.out.println("Argument cannot be negative");
      }
   }
}
```

Factorial recursive

Recursive factorial definition:

$n! = 1 \quad \text{if } n = 0$

$n! = (n-1)!\, n \quad \text{if } n > 0$

```
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.

Ternary conditional operator ? :

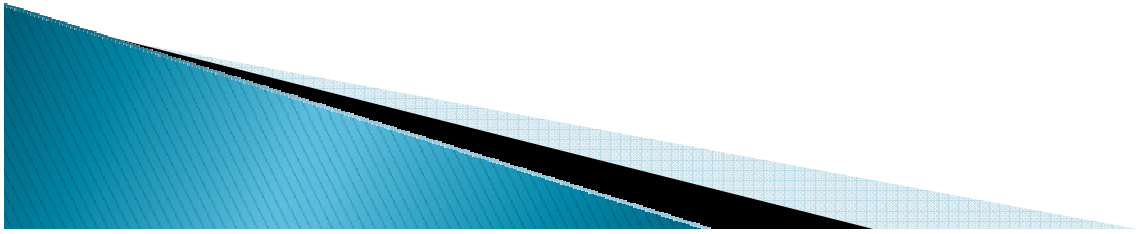
```
1  public class MinTest
2  {
3      public static void main( String [ ] args )
4      {
5          int a = 3;
6          int b = 7;
7
8          System.out.println( min( a, b ) );
9      }
10
11     // Method declaration
12     public static int min( int x, int y )
13     {
14         return x < y ? x : y;
15     }
16 }
```

figure 1.6

Illustration of method declaration and calls

In all your code:

- Write appropriate comments:
  - Javadoc comments for public fields and methods.
  - Explanations of anything else that is not obvious.
- Give explanatory variable and method names:
  - Use name completion in Eclipse, Ctrl-Space, to keep typing cost low and readability high
- Use local variables and static methods (instead of fields and non-static methods) where appropriate.
  - “where appropriate” includes any place where you can’t explicitly justify doing otherwise.
- Use Ctrl-Shift-F in Eclipse to format your code.



Homework due tomorrow

- Let's start together