Tutorial 3: Conditionals and Iteration

COMP 202: Intro to Computing 1

Winter 2009

TA: Robert Rolnick
E-mail: Robert.Rolnick@mail.mcgill.ca

Tutorial 3: Conditionals and Iteration

1. Conditions
2. Conditionals
3. Iteration
4. Practice Programs
5. Addendum

Conditions

- Conditions are expressions that return a boolean (true or false).
- Comparisons and equality tests are the most common conditions. For example (5 < 3) is a condition which will return false.
- Conditions come up frequently when discussing conditional statements, and iteration.
- Strings conditions are done differently than those on fundamental data types such as int, double, etc. An example is coming later.

"The most exciting phrase to hear in science, the one that heralds new discoveries, is not 'Eureka!' but 'That's funny...'"

~Isaac Asimov

Comparison Operators

- The following comparison operators are useful to compare fundamental data types (int, float, double, etc.)

Operator	Name	Example	Returns
==	Equal to	4 == 4	true
!=	Not-Equal to	4 != 4	false
<	Less Than	8 < 8	false
<=	Less Than Or Equal	8 <= 8	true
>	Greater Than	9 > 4	true
>=	Greater Than Or Equal	9 >= 10	false

"If debugging is the process of removing software bugs, then programming must be the process of putting them in."

~Edsger Dijkstra

Comparison Operators (Continued)

- Although you can compare floating point numbers (floats and doubles) with the == and != operators, it is discouraged.
- This is because floating point round off error.
- This small bit of error may cause two numbers to be non-equal when compared, despite being (theoretically) equal based on the operations performed.
- When comparing floats to a specific value, you should do it with a tolerance. (In other words, allow yourself to be within a very small range of the number you want.)

Boolean Manipulation

- The following operators can be used to compare booleans:
  - &&, and operator
  - ||, or operator
  - ==, equality operator
  - !=, in-equality operator

A	B	A && B	A || B	A == B	A != B
true	true	true	true	true	false
true	false	false	true	false	true
false	true	false	true	false	true
false	false	false	false	true	false

- There is also a fifth operator called the xor operator. It uses a ^.
- For booleans ^ does the same thing as !=.
- The xor operator's true purpose is beyond the scope of this class.

"People who deal with bits should expect to get bitten."
~Jon Bentley

Boolean Manipulation (Continued)

- Java does short circuiting of the && and || operators.
- Suppose you are given the conditional expression:

```
if (/* condition 1 */ && /* condition 2 */)
```

- If condition 1 is *false*, then condition 2 will not be evaluated.
- Now consider the following conditional expression:

```
if (/* condition 1 */ || /* condition 2 */)
```

- If condition 1 is *true*, then condition 2 will not be evaluated.
- These properties are helpful when you learn about the *null* value.

Boolean Manipulation (Continued)

- You can also use the ! operator, to toggle a boolean.

A	!A
true	false
false	true

Conditionals

- Conditional statements allow you to control your program's flow.
- Using conditionals, you can define multiple paths in your code, and then choose a path based on the result of a condition.
- Java offers three different types of conditional arguments: if statements, switch statements, and ternary statements.
- Ternary statements are barely used in this course, but they can be very helpful in some circumstances.

"I know it doesn't sound like a big effort, but programmers are really, really lazy, and they like to minimize motion."

~Steve Yegge

Conditionals (Continued)

- Below is a list of keywords related to conditional statements.
  - `break`
  - `case`
  - `default`
  - `else`
  - `if`
  - `switch`
- By the end of this tutorial, you should understand all of them.
- You can also nest conditional statements. That is to say, you can place one conditional statement inside another.

If Statements

- In its simplest form, an *if* statement looks like this:

```
if (/* condition */)
{
  //Condition evaluated to true
}
```

- The code inside the if statement only gets evaluated if the condition gets evaluated to true.

- Any code before or after an if statement gets evaluated regardless of the result of the condition.

"Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away."

~ Antoine de Saint Exupéry

If – Else Statements

- In addition to saying what to do if the condition is true, you can also say what to do if the condition evaluates to false.

- You do this with an *if-else* statement.

```
if (/* condition */)
{
   //Condition evaluated to true
}
else
{
   //Condition evaluated to false
}
```

If – Else If – Else Statements

- You can even define more paths using *else if* statements.

```
if (/* condition 1 */)
{
  //Condition1 evaluated to true
}
else if (/* condition 2 */)
{
  //Condition1 evaluated to false
  //Condition2 evaluated to true
}
else
{
  //Condition1 evaluated to false
  //Condition2 evaluated to false
}
```

If Statements: Tips

- Both *else if* statements and *else* statements are optional.
- You can chain as many *else if* statements as you want.
- You can use at most one *else* statement per if block.
- The *else* statement is always the last one in an if block.
- At each if block, the code will take at most one path. (If you have an else statement, the code is guaranteed to take one path.)
- If execution path takes the first condition that evaluates to true.

If Statements: Tips (Continued)

- You don't need the block (brace brackets) around an if statement! Should they be absent, the first line following the condition is executed as the block. For example:

```
int x = 8;

if ((x % 2) == 0)
   System.out.println("x is even");
else
   System.out.println("x is odd");
```

- This approach is legal but highly discouraged because it leads to hard to find bugs. Also, it may act unexpectedly when nesting ifs.

If Statements: Conditions

- The following is a valid if statement:

```
if (2 < 4)
{
   System.out.println("The if was taken");
}
```

- The following, while valid, is excessive:

```
if ((2 < 4) == true)
{
   System.out.println("The if was taken");
}
```

If Statements: Conditions (Continued)

- Strings cannot be reliably compared with ==
- To compare strings use either the equals() or equalsIgnoreCase() method. Consider the following example:

```
String s1 = "hello world";
String s2 = "HELLO WORLD";

if (s1.equals(s2))
  System.out.println("s1 equals s2");
else if (s1.equalsIgnoreCase(s2))
  System.out.println("s1 and s2 differ in case only");
else
  System.out.println("s1 and s2 are different");
```

- Which output is expected? Test it. Did you get the expected result?

If Statements: Conditions (Continued)

- Be careful, in an if – else if – else block, the code path will take the first condition that it can.
- Consider the following example:

```
String s1 = "hello world";
String s2 = "hello world";

if (s1.equalsIgnoreCase(s2))
  System.out.println("s1 and s2 differ in case only");
else if (s1.equals(s2))
  System.out.println("s1 equals s2");
else
  System.out.println("s1 and s2 are different");
```

- Which output is expected? Test it. Did you get the expected result?

If Statements: Conditions (Continued)

- Be careful with how code is paired when you omit the braces.
- Consider the following example:

```
int x = 1;

if (x > 0)
  if (x > 5)
    x += 5;
else
  x -= 5;

System.out.println(x);
```

- What output is expected? Test it. Did you get the expected result?

If Statements: Bug Fixing 1

- The following code should determine if x is even.
- There is a subtle bug in the following code, can you spot it?

```
public class SubtleBug1
{
  public static void main(String[] args)
  {
    int x = 8;
    if ((x % 2) == 0);
    {
      System.out.println("x is even");
    }
  }
}
```

- Will the code compile? Will it run as desired?

If Statements: Bug Fixing 1 – Solution

- The code will compile, run, and when x is even it'll do what we wanted. When x is odd, though, it'll fail.
- The bug was on this line

```
if ((x % 2) == 0);
```

- The ending semi-colon should not be there.
- It causes the compiler to enter the no brace brackets handling of the if statement.
- The statement it handles is ';' which is equivalent to "do nothing."
- It then executes the block of code, regardless of the if's condition!

If Statements: Bug Fixing 2

- The following code should print whether the if condition was true or false. It contains a subtle bug in this code too, can you spot it?

```
public static void main(String[] args)
{
  if (false) {
    String s = "true";
  } else {
    String s = "false";
  }
  System.out.println("cond was " + s);
}
```

- Will the code compile? Will it run as desired?

If Statements: Bug Fixing 2 – Solution

- The code won't compile. You'll get the following error: "cannot find symbol."
- This is because we are declaring a variable inside the if block, and thus scoping it to inside the block. Don't forget a variable is only "in scope" until the closing brace of the block it was declared in.
- When we try to use it outside of the block, we are trying to read a variable outside of the scope where it is defined.
- One solution is to declare "String s" before the if block, and then assign to it inside the block.

Switch Statements

- Switch statements are useful when you have a finite number of well defined conditions that you want to check for equality.
- Any switch statement can be converted to an if – else if – else block, but the reverse is not true.
- Choosing to use a switch statement is a judgement call, as there is no performance gain in using them in Java.
- Switch statements can lead to cleaner code than if statements, but they have more subtleties as well.

"The competent programmer is fully aware of the limited size of his own skull. He therefore approaches his task with full humility, and avoids clever tricks like the plague."

~Edsger Dijkstra

Switch Statements: Example 1

- A simple switch statement is shown below:

```
int x = 2;

switch (x)
{
   case 1:
      System.out.println("x is one");
      break;
   case 2:
      System.out.println("x is two");
      break;
   default:
      System.out.println("x is not one or two");
      break;
}
```

Switch Statements: Example 2

- You can switch on integers, shorts, bytes, and character literals.

```
char c = 'a';

switch (c)
{
   case 'a':
      System.out.println("The char is a");
      break;
   case 'b':
      System.out.println("The char is b");
      break;
   default:
      System.out.println("The char is not a or b");
      break;
}
```

Switch Statements: Fall Through

- If you don't put the break statement, the code will "fall through" to the next case. For example:

```
int x = 2;
switch (x)
{
   case 2:
   case 3:
   case 5:
   case 7:
      System.out.println("x = prime number <10");
      break;
   default:
      System.out.println("x != prime number <10");
      break;
}
```

Switch Statements: Tips

- Fall through happens so long as there is no break statement, if there is code in a case it will be executed.
- The final break statement can be omitted. Keeping it there helps prevent bugs when modifying your code. Similarly, the default case need not be at the end, but it is customary to place it there.
- In a switch statement, the ordering of the cases is irrelevant.
- Switch statements only do exact matches, and unlike if statements, switch statements go to the “best” hit.
- Case statements end with a colon. They don’t use brace brackets.

Ternary Statements

- The ternary operator is NOT required for this course, as it is a little advanced.
- The ternary operator allows you to do conditional assignment.
- Since it is used for assigning values, it requires outputs for when the condition is true, and when the condition is false.
- A ternary operation is shown below, it uses the ?: operator.

```
(/*cond*/) ? /*cond is true*/ : /*cond is false*/;
```

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

~ Martin Fowler

Ternary Statements: Example

- The following is a sample program using the ternary operator.

```
import java.util.Scanner;

public class MyClass
{
  public static void main(String[] args)
  {
    Scanner keyboard = new Scanner(System.in);
    int x = keyboard.nextInt();
    String s = ((x % 2) == 0) ? "even" : "odd";
    System.out.println("x is " + s);
  }
}
```

Iteration

- Iteration, often called looping, allows you to repeat the execution of your code.
- Loops make code shorter, more concise, and easier to maintain.
- Java offers three different types of loops: *for* loops, *while* loops, and *do – while* loops.
- Choosing your loop's ending conditions wisely! Otherwise, the loop could repeat forever. This is known as an infinite loop, and makes your program hang.

"If you don't have time to do it right, when will you have time to do it over?"
~John Wooden

Iteration (Continued)

- Below is a list of keywords related to iteration.
  - `break`
  - `continue`
  - `do`
  - `for`
  - `while`
- By the end of this tutorial, you should understand all of them.
- Like conditional statements, you can nest iterative code. That is you can place one loop inside another. You can even nest conditional statements and loops inside each other.

For Loops

- For loops are useful when you need to loop over a specific range of numbers by constant increments (or decrements.)

- The template of a for loop is shown below:

```
for (/*start*/; /*condition*/; /*increment*/)
{
  //Code here repeats
}
```

- The condition is the same as for conditional statements.

- The loop repeats so long as the condition is true.

"Any darn fool can make something complex; it takes a genius to make something simple."

~ Pete Seeger

For Loops: Example

- Suppose we wanted to write a program that computes the sum of the all the perfect squares from 0 to 10 (inclusive.)

```
public static void main(String[] args)
{
   int sum = 0;

   for (int i = 0; i <= 10; i++)
   {
      sum += i*i;
   }

   System.out.println("Sum is: " + sum);
}
```

- Have you learned about the ++ operator?

For Loops: Tips

- Like if statements, you don't need the block (brace brackets) around a for loop! Should they be absent, the first line following the condition is executed as the block.

- For example:

```
int sum = 0;

for (int i = 0; i <= 10; i++)
   sum += i*i;

System.out.println("Sum is: " + sum);
```

- Once again, it is discouraged because it leads to hard to find bugs.

For Loops: Bug Fixing 1

- This code should compute the sum the cubes from 0 to 10.
- It contains a subtle bug, can you spot it?

```
public static void main(String[] args)
{
   int sumCubes = 0;

   for (int i = 0; i <= 10; i++)
   {
      sumCubes = i*i*i;
   }

   System.out.println("Sum is: " + sumCubes);
}
```

- Will the code compile? Will it run as desired?

For Loops: Bug Fixing 1 – (Solution)

- The code will compile and run.
- However the bug is with the following line:

```
sumCubes = i*i*i;
```

- Each time through, the loop we are storing the cube of i. Doing this replaces the old value stored in sumCubes.
- We are never accumulating the cubes!

For Loops: Bug Fixing 2

- The following program should output the value of i, as i ranges from 0 to 10 inclusive.
- There is a subtle bug in the following code, can you spot it?

```
public static void main(String[] args)
{
   int i = 5;

   for (i = 0; i <= 10; i++);
   {
      System.out.println("i is: " + i);
   }
}
```

- Will the code compile? Will it run as desired?

For Loops: Bug Fixing 2 – (Solution)

- The code will compile and run.
- Like in the if statement example, the bug is on the following line:

```
for (i = 0; i <= 10; i++);
```

- The ending semi-colon should not be there.
- It causes the compiler to enter the no brace brackets handling of the for loop. It handles the ';' (or "do nothing") statement.
- It then it repeats the "do nothing" loop.
- You'll ultimately print out "i is 11"
- Had "int i" not been declared outside of the loop, we would have had a compile error.

For Loops: Bug Fixing 3

- The following code should count from 0 to 2 in increments of 0.05.
- There is a VERY subtle bug in the following code, can you spot it?

```
public static void main(String[] args)
{
  for (double d = 0; d != 2; d += 0.05)
  {
    System.out.println("d is: " + d);
  }
}
```

- Will the code compile? Will it run as desired?

For Loops: Bug Fixing 3 – (Solution)

- The code will compile and infinitely loop.
- The error is on the following line.

```
for (double d = 0; d != 2; d += 0.05)
```

- Checking the output we get

```
d is: 1.9500000000000001
d is: 2.0000000000000001
d is: 2.0500000000000007
```

- This is due to floating point round off error, which tends to be an advanced topic. Handling it is not typically part of this course.

While Loops

- A sample while loop is shown below

```
while (/* condition */)
{
   //repeats while condition is true
}
```

- There is no semi colon when you initialize the while loop.
- You do not need the brace brackets around the while loop. Should they be absent, the first line following the condition is executed as the block. It is discouraged for the reasons we discusses before.
- Watch out for infinite loops when choosing your condition.

"Get it right. Then get it fast."
~ Steve McConnell

While Loops: Example 1

- The following code counts down from 10 to 1

```
public static void main(String[] args)
{
   int countDown = 10;

   while (countDown > 0)
   {
      System.out.println(countDown);
      countDown--;
   }
}
```

While Loops: Example 2

- Now we count up from 1 to 10

```
public static void main(String[] args)
{
   int countUp = 1;

   while (countUp <= 10)
   {
      System.out.println(countUp);
      countUp++;
   }
   System.out.println(countUp);
}
```

- What do you expect the final println statement to print? Test it. Did it print what you expected?

While Loops: Bug Fixing 1

- The following code should countdown from 99 to 0, showing only odd values. It has a subtle bug, can you spot it?

```
public static void main(String[] args)
{
   int countDown = 99;

   while (countDown != 0)
   {
      System.out.println(countDown);
      countDown -= 2;
   }
}
```

- Will the code compile? Will it run as desired?

While Loops: Bug Fixing 1 – (Solution)

- The code will compile and loop infinitely.
- This is because of an error on the following line:

```
while (countDown != 0)
```

- countDown goes from 1 to -1, and never hits the end condition.

While Loops: Bug Fixing 2

- The following code should countdown from 99 to 0, showing only odd values. It has a subtle bug, can you spot it?

```
public static void main(String[] args)
{
   int countDown = 99;

   while (countDown <= 0)
   {
      System.out.println(countDown);
      countDown -= 2;
   }
}
```

- Will the code compile? Will it run as desired?

While Loops: Bug Fixing 2 – (Solution)

- The code will compile, run, but print nothing.
- This is because of an error on the following line:

```
while (countDown <= 0)
```

- countDown starts at 99, which is NOT less than or equal to 0, and thus the loop never runs.

While Loops: Bug Fixing 3

- The following code should print all the numbers from 0 to 100.
- It has a subtle bug, can you spot it?

```
public static void main(String[] args)
{
   int countUp = 0;

   while (countUp <= 100)
      System.out.println(countUp);
}
```

- Will the code compile? Will it run as desired?

While Loops: Bug Fixing 3 – (Solution)

- The code will compile and run.
- It'll loop infinitely printing 0 every time.
- This is because we are never changing the value of countUp.
- countUp is initialized to 0, and stays there as we loop forever checking to see if countUp <= 100, which is always true.

While Loops: Bug Fixing 4

- The following code should print all the even numbers between 0 and 100 inclusive. It has a subtle bug, can you spot it?

```
public static void main(String[] args)
{
   int countUp = 0;

   while (countUp <= 100);
   {
      System.out.println(countUp);
      countUp += 2;
   }
}
```

- Will the code compile? Will it run as desired?

While Loops: Bug Fixing 4 – (Solution)

- The code will compile and run forever. It'll print nothing during this infinite loop.
- This is because of an error on the following line:

```
while (countUp <= 100);
```

- The ending semi-colon should not be there.
- It causes the compiler to enter the no brace brackets handling of the for loop.
- The statement it handles is ';' which is equivalent to "do nothing."
- It then repeats the "do nothing" loop checking if countUp is less than or equal to 100. countUp never changes, it stays at 0 forever. Thus, the loop runs forever.

Do – While Loops

- Do – While loops are essentially while loops that are guaranteed to run at least once.
- The template of a the loop is shown below.

```
do
{
   //Run once, then repeat if condition is true.
} while (/* condition */);
```

- Note that there is no semi-colon at the end of the do, but there is one after the while.
- The code in the do executes before we check the while condition.

"Nothing is more permanent than a temporary solution."

~ Anonymous

Do – While Loops: Example 1

- The following do – while loop, prints all the even numbers between 0 and 100 inclusive.

```
public static void main(String[] args)
{
   int countUp = 0;

   do
   {
      System.out.println(countUp);
      countUp += 2;
   } while (countUp <= 100);
}
```

Do – While Loops: Example 2

- The following do – while loop, should print the numbers between count and 100 inclusive.

```
int count = 1000;

do {
   System.out.println(count);
   count++;
} while (count <= 100);

System.out.println(count);
```

- What does the output look like?

- What would the output be if a while loop was used instead?

Do – While Loops: Bug Fixing 1

- The following code should countdown from 99 to 0, showing only odd values. It has a subtle bug, can you spot it?

```
public static void main(String[] args)
{
   int countDown = 99;

   do
   {
      System.out.println(countDown);
      countDown -= 2;
   } while (countDown >= 0)
}
```

- Will the code compile? Will it run as desired?

Do – While Loops: Bug Fixing 1 – (Solution)

- The code will not compile!
- The compiler will point out the following error:

```
Foo.java:13: ';' expected
            } while (countDown >= 0)
                                    ^
1 error
```

- That means that the compiler thinks a semi-colon should be placed at the end of line 13 in Foo.java
- Indeed, there should be a semi-colon at the end of the while statement of a do – while loop.

Do – While Loops: Bug Fixing 2

- The following code should countdown from 99 to 0, showing only odd values. It has a subtle bug, can you spot it?

```
public static void main(String[] args)
{
   int countDown = 99;

   do;
   {
      System.out.println(countDown);
      countDown -= 2;
   } while (countDown >= 0);
}
```

- Will the code compile? Will it run as desired?

Do – While Loops: Bug Fixing 2 – (Solution)

- The code will not compile! Instead, it will display eight errors!
- The first of these errors is:

```
Foo.java:7: while expected
  do;
     ^
```

- The compiler is saying that it is expecting a while after the do.
- This makes sense. The compiler is reading our code as the braceless do-while, with the sole statement being “do nothing.”
- To fix this, remove the semi-colon at the end of the do.

Do – While Loops: Bug Fixing 3

- The following code should countdown from 99 to 0, showing only odd values. It has a subtle bug, can you spot it?

```
public static void main(String[] args)
{
   int countDown = 99;

   do
      System.out.println(countDown);
      countDown -= 2;
   while (countDown >= 0);
}
```

- Will the code compile? Will it run as desired?

Do – While Loops: Bug Fixing 3 – (Solution)

- The code will not compile! The compiler will show three errors, the first of which is:

```
HelloWorld.java:8: while expected
    System.out.println(countDown);
                                  ^
```

- This is because the brace brackets were omitted.

- Strictly speaking they need not be there, but then the do – while loop will consist only of the single line between the do and the while.

- Again, this is discouraged for all the reasons already discussed.

Additional Notes

- We have now discussed the three primary forms of loops in java.
- There is technically a fourth looping structure as well. It is known as a “for each” loop, and uses the ‘for’ keyword in a new way.
- Everything the for each loop does, can be done with a regular for loop. The for each loop is just slightly cleaner to read.
- The for each loop iterates through a collection. As such, it won’t make much sense until you’ve seen arrays or lists.

“Computers are useless. They can only give you answers.”
~ Pablo Picasso

Additional Notes: Break

- The break keyword will exit your loop.
- Suppose you were writing code to evaluate an exponent.
- This can be done with a simple for loop.
- Now, suppose we wanted to stop early, to prevent our calculation from exceeding the maximum value that an integer can store.
- This can be done with the break statement, as shown on the following slide.

Additional Notes: Break (Continued)

```
public static void main(String[] args)
{
   int base = 5;
   int exponent = 20;
   int result = 1;

   for (int i = 0; i <= exponent; i++) {
      if ((Integer.MAX_VALUE / base) < result) {
         break;
      } else {
         result *= base;
      }
   }
}
```

Additional Notes: Break (Continued)

- Using a break statement, as shown, is one way of fixing the loop.
- How many other ways can you think of? Can you think of advantages and disadvantages of each?
- Admittedly, the example given was slightly contrived. When might you see break statement in the "real world?"
- What if you're looking through a list for a specific value, and find it? Would you want to continue searching the full list?
- For this course, instead of using a break statement, it is often better to modify the loop condition.

Additional Notes: Continue

- The continue keyword will immediately move your code's execution to the loop's close brace. From there it may continue looping, or determine that it should stop looping.
- The for loop increment WILL happen.
- Before an example though, let's introduce a couple commands.
- charAt() returns the character at a specific point in the string. (Strings start at character 0.)
- length() returns the total number of characters in a string.

Additional Notes: Continue (Continued)

- Can you figure out what the following code does?

```
public static void main(String[] args)
{
   String s = "a man a plan a canal panama";
   int count = 0;

   for (int i = 0; i < s.length(); i++){
      if (s.charAt(i) == ' ') {
         continue;
      }
      count++;
   }
   System.out.println(count);
}
```

- Can you rewrite the code to avoid the continue statement?

Fibonacci Sequence

- The first two Fibonacci numbers are $F_0 = 0$, $F_1 = 1$.
- The next Fibonacci number is found by taken the sum of the previous two. Mathematically this is written as: $F_n = F_{n-1} + F_{n-2}$
- The Fibonacci sequence looks like: { 0, 1, 1, 2, 3, 5, 8, 13, 21, … }
- Write a program the outputs the first 40 Fibonacci numbers, using each of the three loops seen. (You may need to use longs.)
- Later in the course, you'll learn about recursion. At that point test the recursive version against your iterative one. What do you see?

"Software isn't about methodologies, languages, or even operating systems. It is about working applications."

~ Christopher Baus

Primes

- A prime number is defined as an integer greater than 1, which is only divisible by itself and 1.
- For example 2 is prime because 2 is divisible only by itself and 1.
- There are infinitely many primes, the first few are: 2, 3, 5, 7, 11…
- 9 is not prime, because 9 = 3*3.
- Write a program that finds the first 20 primes.

"Process is no substitute for synaptic activity."

~ Jeff DeLuca

High Low Game

- Now let's write a simple game!
- It starts by randomly picking an integer between 1 and 1000.
- It then asks the user to guess the number, saying if their guess is too low or too high.
- The game stops when the user guesses the number.
- Code to pick a random number is given on the next slide.

"Plan to throw one away; you will anyway."
~ Fred Brooks

High Low Game (Continued)

```
import java.util.Random;

public class RandomNumber
{
  public static void main(String[] args)
  {
    Random generator = new Random();
    //pick a random number in the range: [0,1000)
    //meaning 0 is included, but 1000 is NOT.
    int randomNumber = generator.nextInt(1000);
    System.out.println(randomNumber);
  }
}
```

General Assignment Advice & Tips (Repeated)

- Start on the assignments early. Later ones do get harder.
- Ask clarifying questions on MyCourses so everyone can benefit.
- Alternatively, TAs can help 1-on-1 during their office hours.
- The warm-up questions cover the same material as the graded problems. As such, they serve as excellent building blocks.
- In order to grade your code, TAs need to read it. Clear variable names, proper structure, and comments lead to happier TAs.

"On two occasions, I have been asked [by members of Parliament], 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able to rightly apprehend the kind of confusion of ideas that could provoke such a question." ~ *Charles Babbage*

Specific Tips for Assignment #2

- The second assignment deals with conditionals and iteration.
- Be very careful when writing loop boundaries. The most common mistakes are infinite loops and off by one errors.
- Read the whole assignment before starting, and plan your code before writing it. This can help prevent band-aid bug fixing.
- Some of the ideas reappear between problems. Therefore, think through your solutions carefully. Even if you can't reuse code directly, similar problem solving strategies may apply.

"Debugging is anticipated with distaste, performed with reluctance, and bragged about forever."

~ Anonymous

Floating Point Rounding Error

- Consider the following loop:

```
for(double d = 0.0; d <= 1.0; d += 0.05)
  System.out.println(d);
```

- It gives the following output:

```
0.0                  0.35                 0.7000000000000001
0.05                 0.3999999999999997   0.7500000000000001
0.1                  0.4499999999999996   0.8000000000000002
0.15000000000000002  0.4999999999999994   0.8500000000000002
0.2                  0.5499999999999999   0.9000000000000002
0.25                 0.6                  0.9500000000000003
0.3                  0.65
```

- Some values are too large, others too small. 1.0 is never printed!

"It is easier to write an incorrect program than understand a correct one."
~ Alan Perlis

Avoiding Floating Point Error: Integers

- You can fix the floating point error by using integer values instead of floating point numbers as your loop condition.

- For instance, if you only care about two decimal places, you can use numbers 100 times larger than you actually need. Then, when necessary, divide by 100.0 to get the correct decimal value.

```
for(int i = 0; d <= 100; d++)
  System.out.println(i/100.0);
```

- Alternatively, you can pre-compute how many times to loop. Using an integer as your loop boundary, loop the pre-computed number of times. Other information can be maintained inside of the loop.

Avoiding Floating Point Error: Tolerances

- You can also use tolerance in your loop.
- A typical tolerance is half the increment value.
- Instead of using the following loop:

```
for(double d = 0.0; d <= 1.0; d += 0.05)
{
}
```

- Use this one:

```
for(double d = 0.0; d <= 1.025; d += 0.05)
{
}
```

- Note that 1.025 equals 1.0 + 0.05 / 2

Questions?


HOW'S THE DEBUGGING GOING?
JORGE CHAM © 2005
www.phdcomics.com
CLICK.
COMPILE COMPILE
A-HA! YEAH!! HA! WHO'S YOUR MOMMY!? WHO'S YOUR MOMMY!? GOTCHA! WHOOO..!
SUCCESS.
I TAKE IT YOU FIXED THE BUG IN YOUR CODE?
I SQUISHED IT WITH A TEN-TON HAMMER I FIXED IT..! he he...