Floating Point Anomalies

- Anomalous floating point values:
  - Undefined, such as 0.0/0.0:
    - 0.0/0.0 produces result NaN (Not a Number)
    - Any operation involving NaN produces NaN as result
    - Two NaN values cannot be equal
    - Check if number is NaN by using methods:
      - Double.isNaN(double d) or Float.isNAN(float f)
      - Methods return boolean which is true if argument is NaN
  - Overflow, such as 1.0/0.0:
    - 1.0/0.0 produces result Infinity
    - Same rules, results as for NaN:
      - Double.isInfinite(double d)
  - Underflow, when result is smaller than smallest possible number we can represent (absolute value)
    - Complex condition to detect, usually get zero result
Range of double precision numbers

-1.8\times 10^{308} \quad -4.9\times 10^{-324} \quad 0 \quad +4.9\times 10^{-324} \quad +1.8\times 10^{308}

Overflow \quad Underflow \quad Overflow

Example

```java
public class NaNTest {
    public static void main(String[] args) {
        double a=0.0, b=0.0, c, d;
        c= a/b;
        System.out.println("c: " + c);
        if (Double.isNaN(c))
            System.out.println("  c is NaN");
        d= c + 1.0;
        System.out.println("d: " + d);
        if (Double.isNaN(d))
            System.out.println("  d is NaN");
        if (c == d)
            System.out.println("Oops");
        else
            System.out.println("NaN != NaN");
        double e= 1.0, f;
        f= e/a;
        System.out.println("f: " + f);
        if (Double.isInfinite(f))
            System.out.println("  f is infinite");
    }
}
```
Example

public class NaNTest {
    public static void main(String[] args) {
        double a = 0.0, b = 0.0, c, d;
        c = a / b;
        System.out.println("c: " + c); // c: NaN
        if (Double.isNaN(c))
            System.out.println("  c is NaN"); // c is NaN
        d = c + 1.0;
        System.out.println("d: " + d); // d: NaN
        if (Double.isNaN(d))
            System.out.println("  d is NaN"); // d is NaN
        if (c == d)
            System.out.println("Oops");
        else
            System.out.println("NaN != NaN"); // NaN != NaN
        double e = 1.0, f;
        f = e / a;
        System.out.println("f: " + f); // f: Infinity
        if (Double.isInfinite(f))
            System.out.println("  f is infinite"); // f is infinite
    }
}

Doubles Are Bad Loop Counters

// Suppose we have a stepper motor we want to move from
// x= 0 to x= 10 in increments of 0.2

public class Counter {
    public static void main(String[] args) {
        int i = 0;
        double x = 0.0;
        while (x < 10.0) {
            x += 0.2;
            i++;
            if (i % 10 == 0 || i >= 48)
                System.out.println("x: " + x + " i: " + i);
        }
    }
}
Doubles Are Bad Loop Counters

i : 10 x : 1.9999999999999998
i : 20 x : 4.0000000000000001
i : 30 x : 6.0000000000000003
i : 40 x : 8.0000000000000004
i : 48 x : 9.5999999999999998
i : 49 x : 9.7999999999999997
i : 50 x : 9.9999999999999996
i : 51 x : 10.1999999999999996

Notice accumulating, increasing error.
Don’t use floats or doubles as loop counters

We went one iteration too many

Exercise

- Create a class InverseTest. In main():
  - Set xStart= 0.0, xEnd= 2.0, deltax= 0.1
  - Write a ‘for’ loop on x from xStart to xEnd, incrementing x by deltax each time
    - Use double TOLERANCE = 1E-14 to terminate the loop at the correct point. Without TOLERANCE, it won’t.
  - Output x
  - Compute and output 1/(xEnd - x)
  - See next slide for some of the code
- What should happen at the end of the loop?
  - Does Java catch the zero divide?
- If you have time:
  - Implement this with an int loop counter
  - Does this necessarily fix all the problems?
Exercise

```java
public class InverseTest {
    public static void main(String[] args) {
        double xStart= 0.0, xEnd= 2.0, deltax= 0.1;
        final double TOLERANCE= 1E-14;
        for (…)   {  // Your code here
            // Loop on x, which goes from xStart to xEnd
            //   in steps of deltax
            // Output x
            // Compute and output 1/(xEnd-x)
            }
    }
}
```

Numerical Problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>Integer</th>
<th>Float, double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero divide</td>
<td>Program terminates (throws an exception)</td>
<td>Infinity</td>
</tr>
<tr>
<td>0/0</td>
<td>Program terminates (throws an exception)</td>
<td>NaN (not a number)</td>
</tr>
<tr>
<td>Overflow</td>
<td>No warning. Program gives wrong results.</td>
<td>Infinity</td>
</tr>
<tr>
<td>Underflow</td>
<td>Not possible</td>
<td>No warning, set to 0 usually</td>
</tr>
<tr>
<td>Rounding, accumulation errors</td>
<td>Not possible</td>
<td>No warning. Program gives wrong results.</td>
</tr>
</tbody>
</table>

Common, “bad news” cases
More on Control Structures

• Three control structures in Java, or any language:
  – Sequence: execute next statement
    • This is default behavior
  – Branching: if, else statements
    • If, else are the primary construct used
    • Switch statement used if many choices
  – Iteration: while, do, for loops
    • Additional constructs exist to terminate loops 'prematurely'

Terminating Iteration: Break

• Break statement in for, while or do-while loops transfers control to statement immediately after end of loop

```java
public class BreakTest {
    public static void main(String[] args) {
        for (int i = 0; i < 6; i++) {
            if (i >= 3)
                break;  // End loop
            System.out.println(i + i);
        }
        System.out.println("Done");
    }
    // What will this print?
    // If "break" in inner, nested loop, control is
    // transferred to the outer loop
```
Terminating Iteration: Continue

- Continue statement jumps to end of loop but continues looping

```java
public class ContinueTest {
    public static void main(String[] args) {
        for (int i = 0; i < 6; i++) {
            if (i < 4) continue;  // Skip rest of loop
            System.out.println("i: "+i);
        }
        System.out.println("Done");
    }
}
```

// What will this print?
// If "continue" in inner, nested loop, control stays
// in inner loop

Control exercise

- Write a class LoopExercise:
  - main() method has:
    - Loop over int i going from 0 through 8
      - Make j = i^2-5
      - If j negative, skip the rest of the loop
      - Find s= square root of j (use Math.sqrt(j));
      - If s > 4, end the loop
      - Output i, j and s to see what’s happening
  - Print “Done” at the end of the program
  - This is characteristic of, e.g., gearbox design problem:
    - Integer number of teeth
    - Double diameter
    - Minima and maxima for gear ratios, rpms, etc.
    - Loop to find feasible ones (skip rest of loop if infeasible)
    - If feasible, search for best (end the loop when found)
Java Methods

• Methods are discrete units of behavior
  – You’ve already used some:
    • JOptionPane.showInputDialog()
    • Math.sqrt()
    • System.out.println()
  – You’ll write your own for the rest of the term, as part of classes
  – Right now, you are writing classes but they only have a main() method and they create no objects
  – We’ll write additional methods in our classes
  – (And then create objects that have methods)
  – For now, our methods will have the keywords public static in them
    • Treat them as an incantation for this and the next lecture

Java Methods

• Methods are the interface or communications between program components
  – They provide a way to invoke the same operation from many places in your program, avoiding code repetition
  – They hide implementation details from the component using the method
  – Variables defined within a method are not visible to users of the method; they have local scope within the method
  – The method cannot see variables in the component that calls it either. There is logical separation between the two, which avoids conflicts in variable names
Method example

```java
public class MethodExample {
    public static void main(String[] args) {
        double boxWeight= 50;
        double boxCube= 10;
        String boxID= "Box A";
        double density= getDensity(boxWeight, boxCube);
        System.out.println("Density: "+ density);
        printBox(boxWeight, boxCube);  // Prints density 2nd time
    }
    public static double getDensity(double bw, double bc) {
        double result= bw/bc;  // 'result' could be 'density'
        return result;
    }
    public static void printBox(double w, double c) {
        System.out.println("Box weight: "+w+" cube: "+c);
        System.out.println(" Density: "+getDensity(w,c));
        // System.out.println(" ID: "+boxID);  // No access to ID
        // Won't compile!
    }
}
```

Passing Arguments

```
main(...) {  
    double boxWeight= 50;
    double boxCube= 10;
    String boxID= "Box A";
    double density= getDensity (boxWeight, boxCube);
    
```

Communication only via arg list, return value

Arguments matched by position

Data type, meaning must match

Explore with debugger