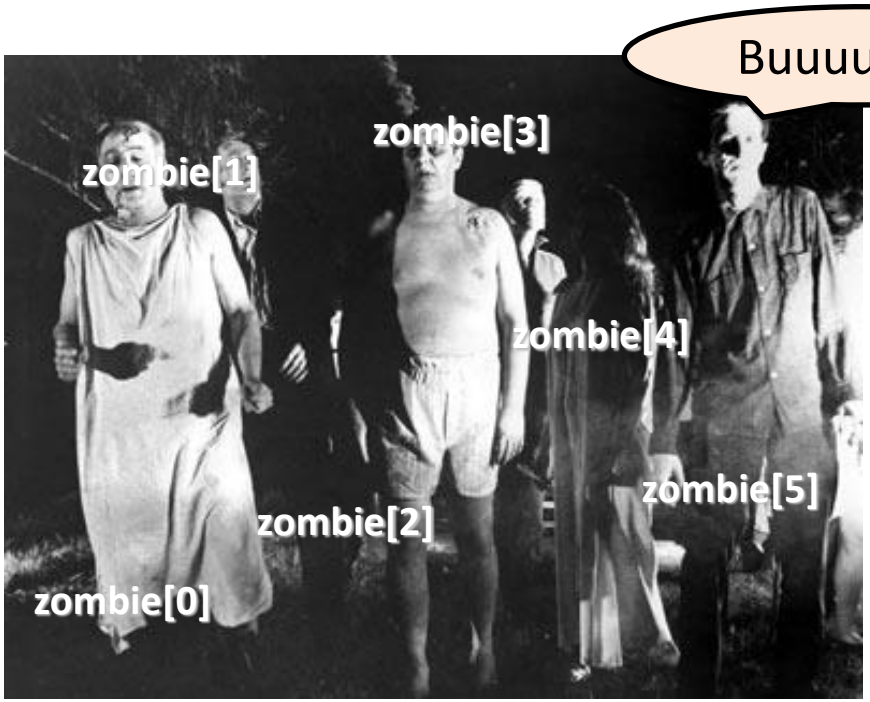
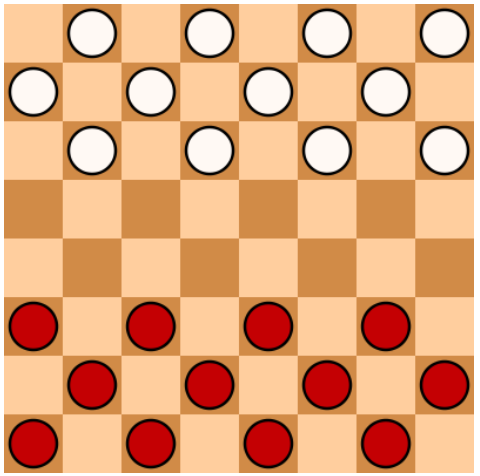
More on variables, arrays, debugging



Overview

- Variables revisited
  - Scoping
- Arrays revisited
  - Multidimensional arrays
- Debugging
  - Tip and tricks to help you keep your sanity



Variable scoping

- Variables live within their curly braces
  - Once curly block finishes, variable is gone!

```
public class DoStuff
{
   public static void main(String [] args)
   {
      int x = 0;
      for (int y = 0; y < 5; y++)
      {
         x = x + y;
      }
      x = x * y;
   }
}
```

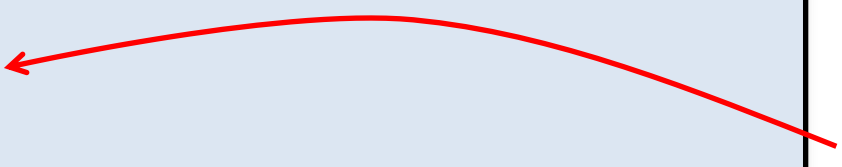
y only lives in the for-loop

y is undefined, this won't compile!

Variable scoping

- You can declare and reuse same name again
  - But only after no longer "in scope"

```
public class DoStuff
{
   public static void main(String [] args)
   {
      int x = 0;
      for (int y = 0; y < 5; y++)
      {
         x = x + y;
      }
      int y = 1;
      x = x * y;
   }
}
```



This fixes the compile error (though doesn't really do anything useful).

Arrays revisited

- Arrays
  - Store a bunch of values under one name
  - Declare and create in one line:

```
int N = 8;
int [] x = new int[10];
double [] speeds = new double[100];
String [] names = new String[N];
```

  - To get at values, use name and index between []:

```
int sumFirst2 = x[0] + x[1];
speeds[99] = speeds[98] * 1.1;
System.out.println(names[0]);
```

  - Array indexes start at 0!

Arrays revisited

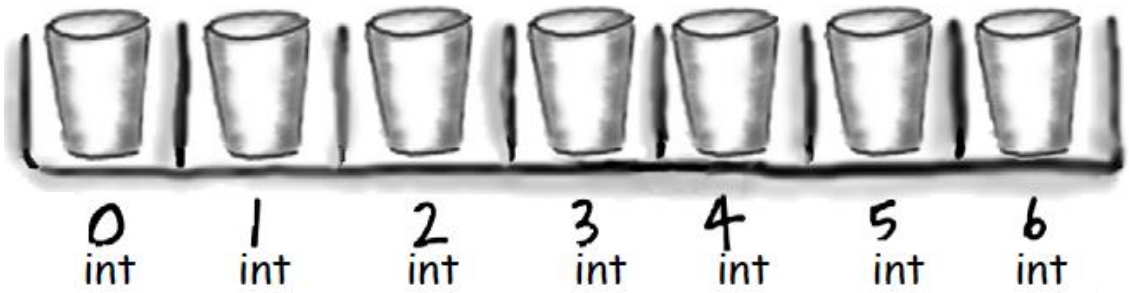
- Arrays
  - You can just declare an array:

```
int [] x;
```

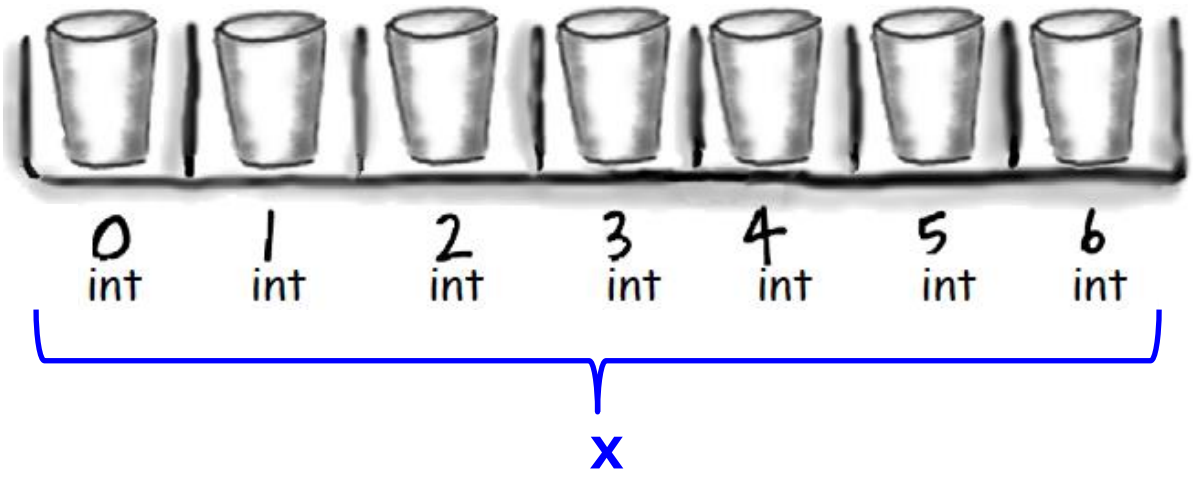
  - But x is not very useful until you "new" it:

```
int [] x;
x = new int[7];
```

  - new creates the memory for the slots
    - Each slot holds an independent int value
    - Each slot initialized to default value for type

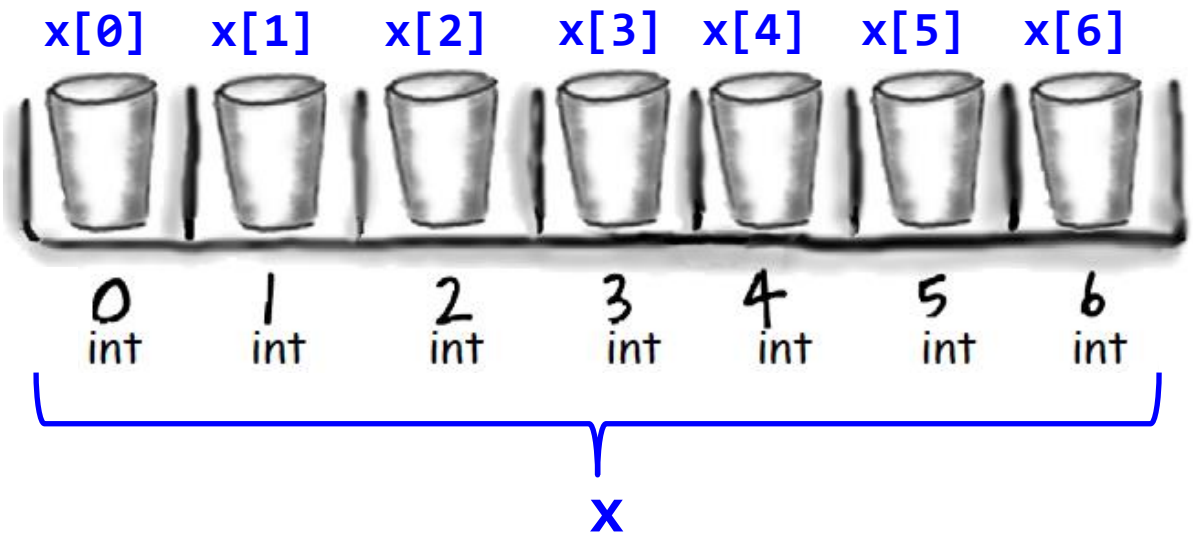


Arrays revisited



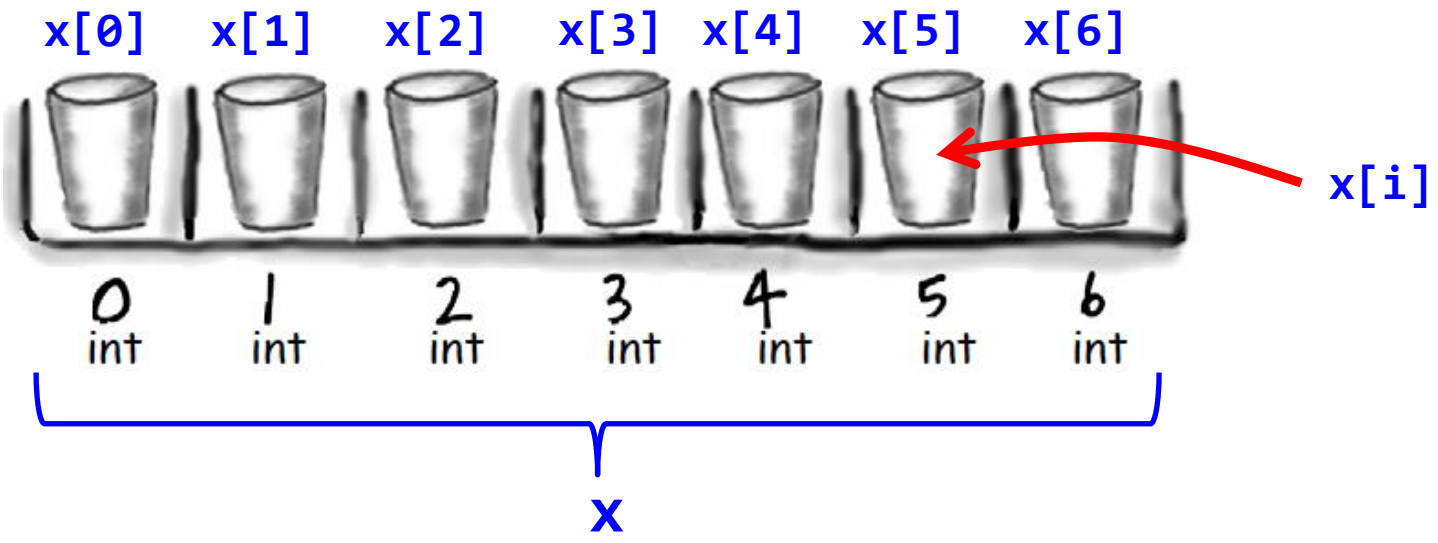
- Variable x refers to the whole set of slots
- You can't use the variable x by itself for much
- Except for finding out the number of slots: `x.length`

Arrays revisited



- `x[0],x[1],…,x[6]` refers to value at a particular slot
- `x[-1]` or `x[7]` = `ArrayIndexOutOfBoundsException`

Arrays revisited

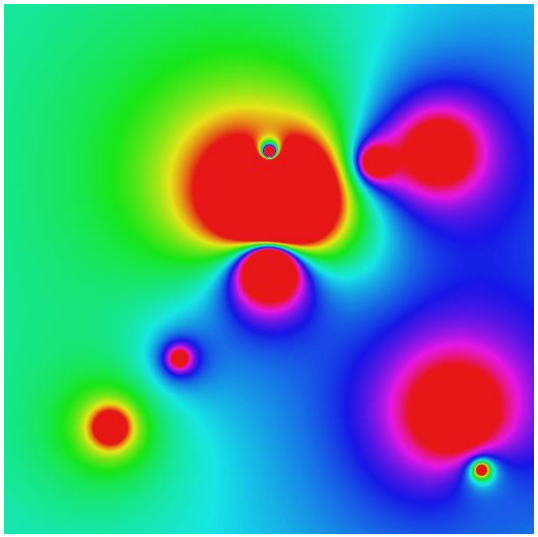


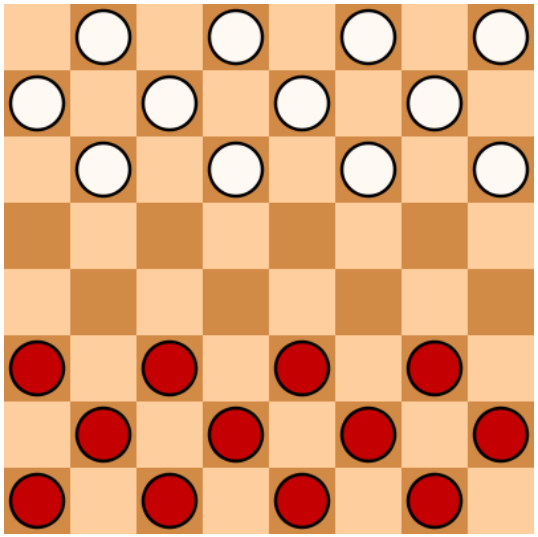
- `x[i]` refers to the value at a slot, but the slot index is determined by variable `i`
  - If i = 0 then `x[0]`, if i = 1 then `x[1]`, etc.
- Whatever inside [] must be an int
- Whatever inside [] must be in 0 to `x.length - 1` (inclusive)

Two dimensional array examples

- Two dimensional arrays
  - Tables of hourly temps for last week
  - Table of colors for each pixel of a 2D image
  - Table storing piece at each position on a checkerboard

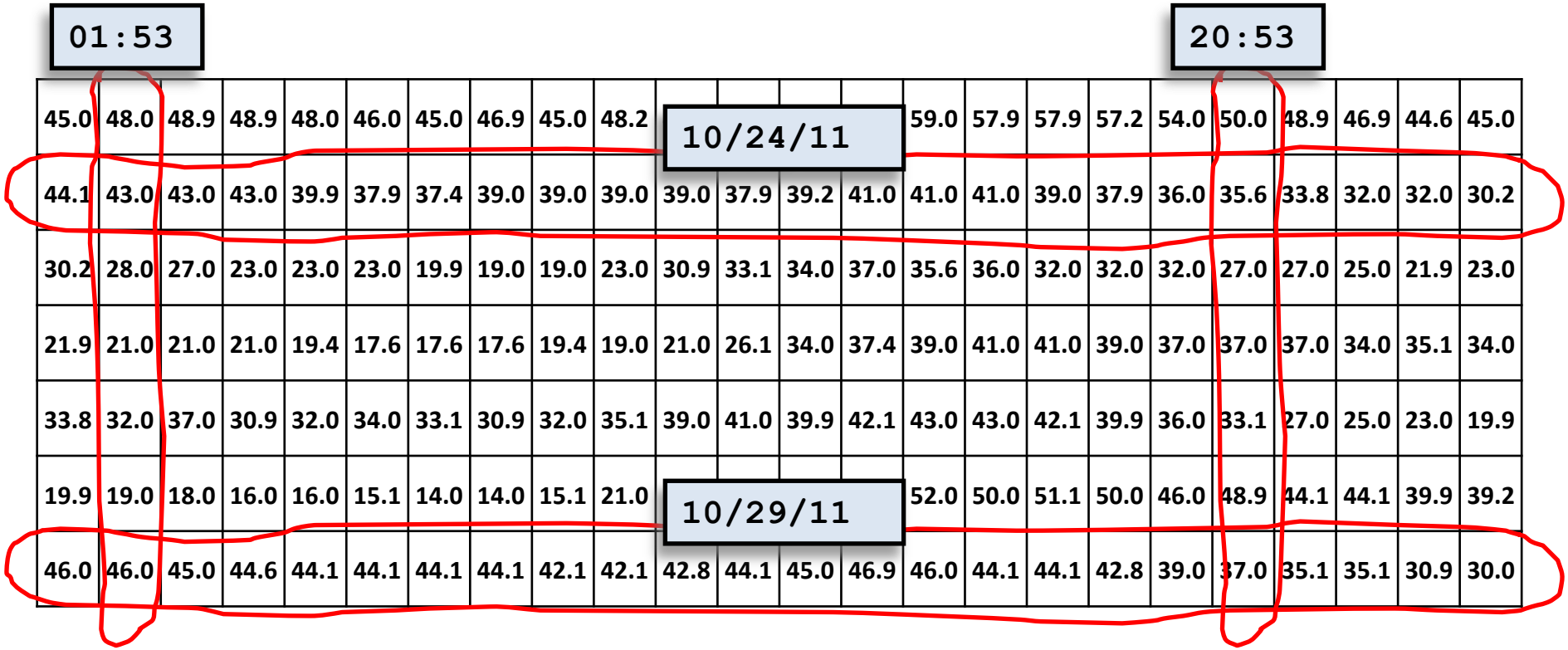
0h	1h	…	23h
32.5	30.0		45.6
…			
59.5	62.1	…	60.0
60.7	61.8	…	70.5
62.6	62.0	…	68.0





Weather data

- Goal: Read in hourly temp data for last week
  - Each row is a day of the week
  - Each column is a particular hour of the day



45.0	48.0	48.9	48.9	48.0	46.0	45.0	46.9	45.0	48.2					59.0	57.9	57.9	57.2	54.0	50.0	48.9	46.9	44.6	45.0
44.1	43.0	43.0	43.0	39.9	37.9	37.4	39.0	39.0	39.0	39.0	37.9	39.2	41.0	41.0	41.0	39.0	37.9	36.0	35.6	33.8	32.0	32.0	30.2
30.2	28.0	27.0	23.0	23.0	23.0	19.9	19.0	19.0	23.0	30.9	33.1	34.0	37.0	35.6	36.0	32.0	32.0	32.0	27.0	27.0	25.0	21.9	23.0
21.9	21.0	21.0	21.0	19.4	17.6	17.6	17.6	19.4	19.0	21.0	26.1	34.0	37.4	39.0	41.0	41.0	39.0	37.0	37.0	37.0	34.0	35.1	34.0
33.8	32.0	37.0	30.9	32.0	34.0	33.1	30.9	32.0	35.1	39.0	41.0	39.9	42.1	43.0	43.0	42.1	39.9	36.0	33.1	27.0	25.0	23.0	19.9
19.9	19.0	18.0	16.0	16.0	15.1	14.0	14.0	15.1	21.0					52.0	50.0	51.1	50.0	46.0	48.9	44.1	44.1	39.9	39.2
46.0	46.0	45.0	44.6	44.1	44.1	44.1	44.1	42.1	42.1	42.8	44.1	45.0	46.9	46.0	44.1	44.1	42.8	39.0	37.0	35.1	35.1	30.9	30.0

Two dimensional arrays

- Declaring and creating
  - Like 1D, but another pair of brackets:

```
final int DAYS  = 7;
final int HOURS = 24;
double [][] a = new double[DAYS][HOURS];
```

- Accessing elements
  - To specify element at the i^{th} row and j^{th} column:

```
a[i][j]
```

a[0][0]	a[0][1]	a[0][2]	...	a[0][22]	a[0][23]
a[1][0]	a[1][1]	a[1][2]	…	a[1][22]	a[1][23]
…	…	…	…	….	…
a[6][0]	a[6][1]	a[6][2]	…	a[6][22]	a[6][23]

Temperature on second day of data, last hour of day

Reading temperature data

- Initialize all elements of our 2D array
  - Nested loop reading in each value from `StdIn`
  - Find weekly max and min temp

```
final int DAYS  = 7;
final int HOURS = 24;
double [][] a = new double[DAYS][HOURS];
double min = Double.POSTIVE_INFINITY;
double max = Double.NEGATIVE_INFINITY;

for (int row = 0; row < DAYS; row++)
{
   for (int col = 0; col < HOURS; col++)
   {
      a[row][col] = StdIn.readDouble();
      min = Math.min(min, a[row][col]);
      max = Math.max(max, a[row][col]);
   }
}
System.out.println("min = " + min + ", max = " + max);
```

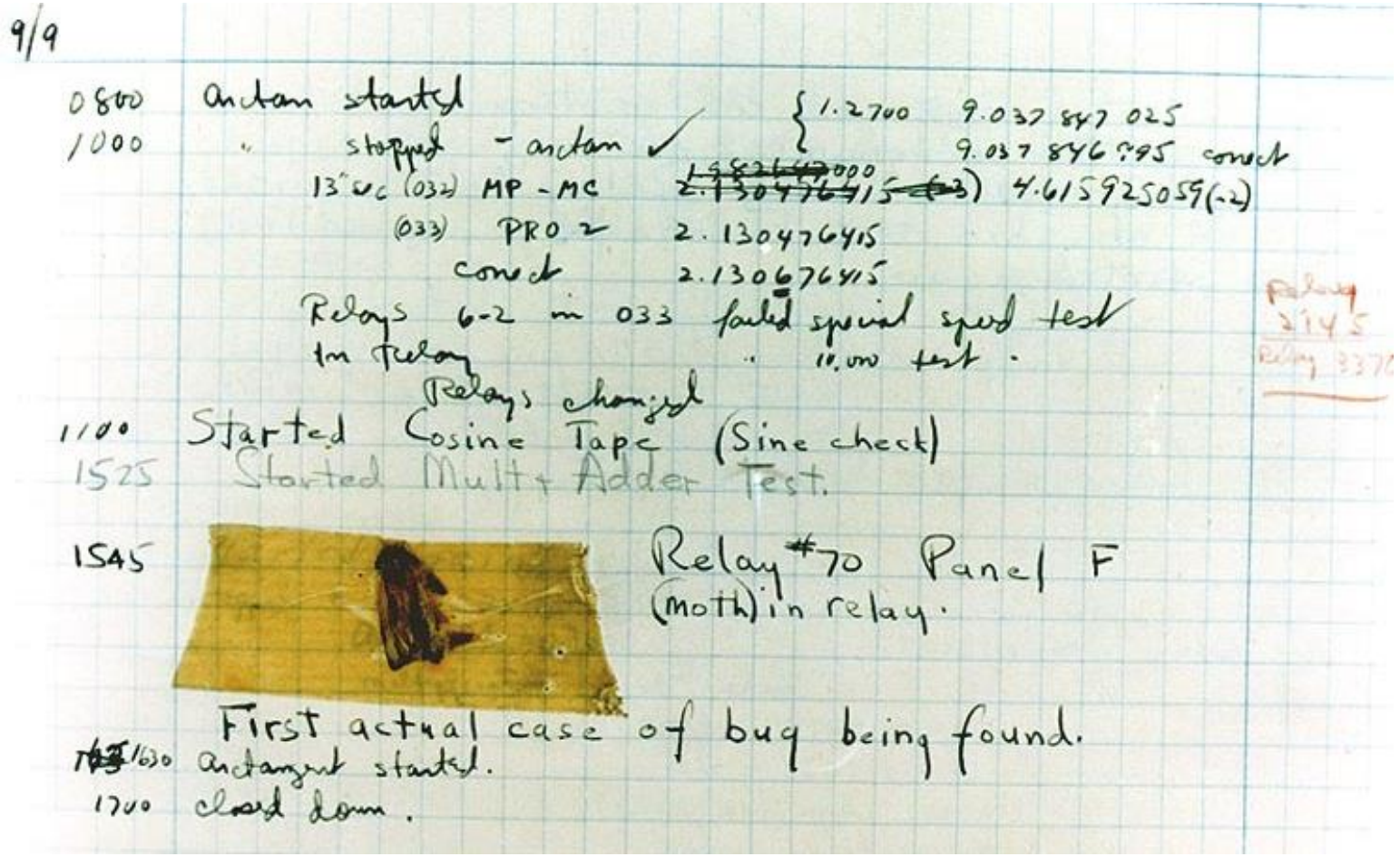
Start the min at a really high temp.

Start the max at a really low temp.

The new min temp is either the current min or the new data point.

Debugging

- Majority of program development time:
  - Finding and fixing mistakes! a.k.a. bugs
  - It's not just you: bugs happen to all programmers



Debugging

- Computers can help find bugs
  - But: computer can't automatically find all bugs!
- Computers do exactly what you ask
  - Not necessarily what you want
- There is always a logical explanation!
  - Make sure you saved & compiled last change

"As soon as we started programming, we found out to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs."
-Maurice Wilkes

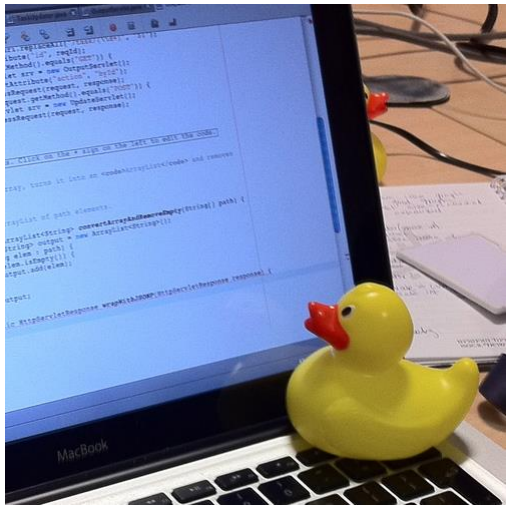
"There has never been an unexpectedly short debugging period in the history of computers."
-Steven Levy

Preventing bugs

- Have a plan
  - Write out steps in English before you code
  - Write comments first before tricky bits
- Use good coding style
  - Good variable names
    - "Name variables as if your first born child"
    - If variable is called `area` it should hold an area!
  - Split complicated stuff into manageable steps
  - ()'s are free, force order of operations you want
- Carefully consider loop bounds
- Listen to Eclipse (IDE) feedback

Finding bugs

- How to find bugs
  - Add debug print statements
    - Print out state of variables, loop values, etc.
    - Remove before submitting
  - Use debugger in your IDE
    - Won't work if using file redirection
  - Talk through program line-by-line
    - Explain it to a:
      - Programming novice
      - Rubber duckie
      - Teddy bear
      - Potted plant
      - …



Debugging example

- Problem:
  - For integer N > 1, compute its *prime factorization*
    - $98 = 2 \times 7^2$
    - $17 = 17$
    - $154 = 2 \times 7 \times 11$
    - $16{,}562 = 2 \times 7^2 \times 13^2$
    - $3{,}757{,}208 = 2^3 \times 7\ 13^2 \times 397$
    - $11{,}111{,}111{,}111{,}111{,}111 = 2{,}071{,}723 \times 5{,}363{,}222{,}357$
  - Possible application: Break RSA encryption
    - Factor 200-digit numbers
    - Used to secure Internet commerce

A simple algorithm

- Problem:
  - For integer N > 1, compute its *prime factorization*
- Algorithm:
  - Starting with i=2
    - Repeatedly divide N by i as long as it evenly divides, output i every time it divides
  - Increment i
  - Repeat

Example run

i	N	Output
2	16562	2
3	8281	
4	8281	
5	8281	
6	8281	
7	8281	7 7
8	169	
9	169	
10	169	
11	169	
12	169	
13	169	13 13
14	1	
…	1	

Buggy factorization program

```
public class Factors
{
   public static void main(String [] args)
   {
      long n = Long.parseLong(args[0])
      for (i = 0; i < n; i++)
      {
         while (n % i == 0)
            System.out.print(i + " ")
            n = n / i
      }
   }
}
```

This program has many bugs!

Debugging: syntax errors

```
public class Factors
{
   public static void main(String [] args)
   {
      long n = Long.parseLong(args[0]);
      for (int i = 0; i < n; i++)
      {
         while (n % i == 0)
            System.out.print(i + " ");
            n = n / i;
      }
   }
}
```

- Syntax errors
  - Illegal Java program
  - Usually easily found and fixed

Debugging: semantic errors

```
public class Factors
{
   public static void main(String [] args)
   {
      long n = Long.parseLong(args[0]);
      for (int i = 0; i < n; i++)
      {
         while (n % i == 0)
            System.out.print(i + " ");
            n = n / i;
      }
   }
}
```

Need to start at 2 since 0 and 1 cannot be factors.

```
% java Factors 98
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at Factors.main(Factors.java:8)
```

- Semantic error
  - Legal but wrong Java program
  - Run program to identify problem

Debugging: semantic errors

```
public class Factors
{
   public static void main(String [] args)
   {
      long n = Long.parseLong(args[0]);
      for (int i = 2; i < n; i++)
      {
         while (n % i == 0)
            System.out.print(i + " ");
            n = n / i;
      }
   }
}
```

Indentation implies a block of code, but without braces, it's not.

```
% java Factors 98
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 ...
```

Debugging: even more problems

```
public class Factors
{
   public static void main(String [] args)
   {
      long n = Long.parseLong(args[0]);
      for (int i = 2; i < n; i++)
      {
         while (n % i == 0)
         {
            System.out.print(i + " ");
            n = n / i;
         }
      }
   }
}
```

```
% java Factors 98
2 7 7 %
```

Need newline

```
% java Factors 5
```

No output???

```
% java Factors 6
2 %
```

Missing the 3???

Debugging: adding trace print statement

```
public class Factors
{
   public static void main(String [] args)
   {
      long n = Long.parseLong(args[0]);
      for (int i = 2; i < n; i++)
      {
         while (n % i == 0)
         {
            System.out.println(i + " ");
            n = n / i;
         }
         System.out.println("TRACE " + i + " " + n);
      }
   }
}
```

```
% java Factors 5
TRACE 2 5
TRACE 3 5
TRACE 4 5

% java Factors 6
2
TRACE 2 3
```

i for-loop should go up to n!

Success?

```
public class Factors
{
   public static void main(String [] args)
   {
      long n = Long.parseLong(args[0]);
      for (int i = 2; i <= n; i++)
      {
         while (n % i == 0)
         {
            System.out.print(i + " ");
            n = n / i;
         }
      }
      System.out.println();
   }
}
```

Fixes the "off-by-one" error in the loop bounds.

Fixes the lack of line feed problem.

```
% java Factors 5
5

% java Factors 6
2 3

% java Factors 98
2 7 7

% java Factors 3757208
2 2 2 7 13 13 397
```

Except for really big numbers...

```
public class Factors
{
   public static void main(String [] args)
   {
      long n = Long.parseLong(args[0]);
      for (int i = 2; i <= n; i++)
      {
         while (n % i == 0)
         {
            System.out.print(i + " ");
            n = n / i;
         }
      }
      System.out.println();
   }
}
```

```
% java Factors 11111111
11 73 101 137

% java Factors 11111111111
21649 51329

% java Factors 111111111111
11 239 4649 909091

% java Factors 1111111111111111
2071723 -1 -1 -1 -1 -1 -1 -1 ...
```

Correct, but too slow

```
public class Factors
{
   public static void main(String [] args)
   {
      long n = Long.parseLong(args[0]);
      for (long i = 2; i <= n; i++)
      {
         while (n % i == 0)
         {
            System.out.print(i + " ");
            n = n / i;
         }
      }
      System.out.println();
   }
}
```

```
% java Factors 11111111
11 73 101 137

% java Factors 11111111111
21649 51329

% java Factors 111111111111
11 239 4649 909091

% java Factors 1111111111111111
2071723 5363222357
```

Faster version

```
public class Factors
{
   public static void main(String [] args)
   {
      long n = Long.parseLong(args[0]);
      for (long i = 2; i*i <= n; i++)
      {
         while (n % i == 0)
         {
            System.out.print(i + " ");
            n = n / i;
         }
      }
      System.out.println();
   }
}
```

Missing last factor

```
% java Factors 98
2 7 7

% java Factors 11111111
11 73 101

% java Factors 11111111111
21649

% java Factors 111111111111
11 239 4649

% java Factors 1111111111111111
2071723
```

Fixed faster version

```
public class Factors
{
   public static void main(String [] args)
   {
      long n = Long.parseLong(args[0]);
      for (long i = 2; i*i <= n; i++)
      {
         while (n % i == 0)
         {
            System.out.print(i + " ");
            n = n / i;
         }
      }
      if (n > 1)
         System.out.println(n);
      else
         System.out.println();
   }
}
```

Handles the "corner case"

Need special case to print biggest factor (unless it occurs more than once)

```
% java Factors 98
2 7 7

% java Factors 11111111
11 73 101 137

% java Factors 11111111111
21649 513239

% java Factors 11111111111111
11 239 4649 909091

% java Factors 1111111111111111
2071723 5363222357
```

Factors: analysis

- How large an integer can I factor?

```
% java Factors 3757208
2 2 2 7 13 13 397

% java Factors 92011111169755555703
92011111169755555703
```

digits	(i <= n)	(i*i <= n)
3	instant	instant
6	0.15 seconds	instant
9	77 seconds	instant
12	21 hours *	0.16 seconds
15	2.4 years *	2.7 seconds
18	2.4 millennia *	92 seconds

* estimated

Incremental development

- Split development into stages:
  - Test thoroughly after each stage
    - Don't move on until it's working!
    - Bugs are (more) isolated to the part you've just been working on
    - Prevents confusion caused by simultaneous bugs in several parts

Summary

- Variables
  - Live within their curly braces
- Arrays
  - Hold a set of independent values of same type
  - Access single value via index between []'s
- Debugging
  - Have a plan before coding, use good style
  - Learn to trace execution
    - On paper, with print statements, using the debugger
  - Explain it to a teddy bear
  - Incremental development