Multidimensional Arrays; Object-Oriented Design

Outline

- Arrays Revisited
  - Multidimensional arrays
- Object-Oriented Design
  - Data encapsulation
  - Checking for equality

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

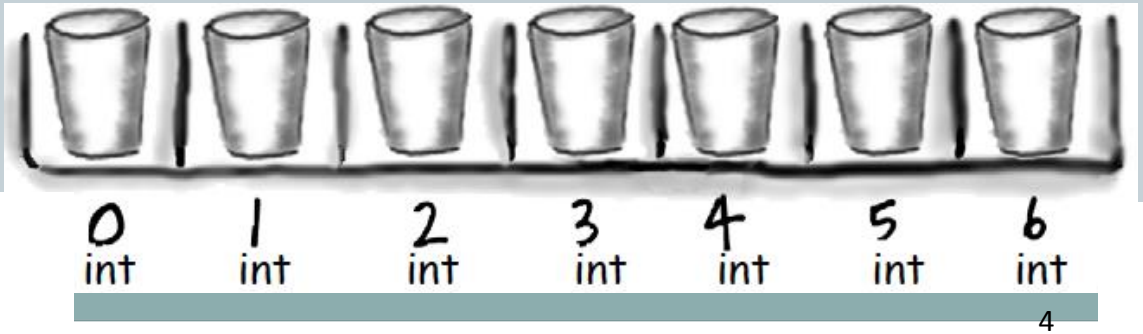
- Arrays
  - You can just declare an array:

```
int [] x;
```

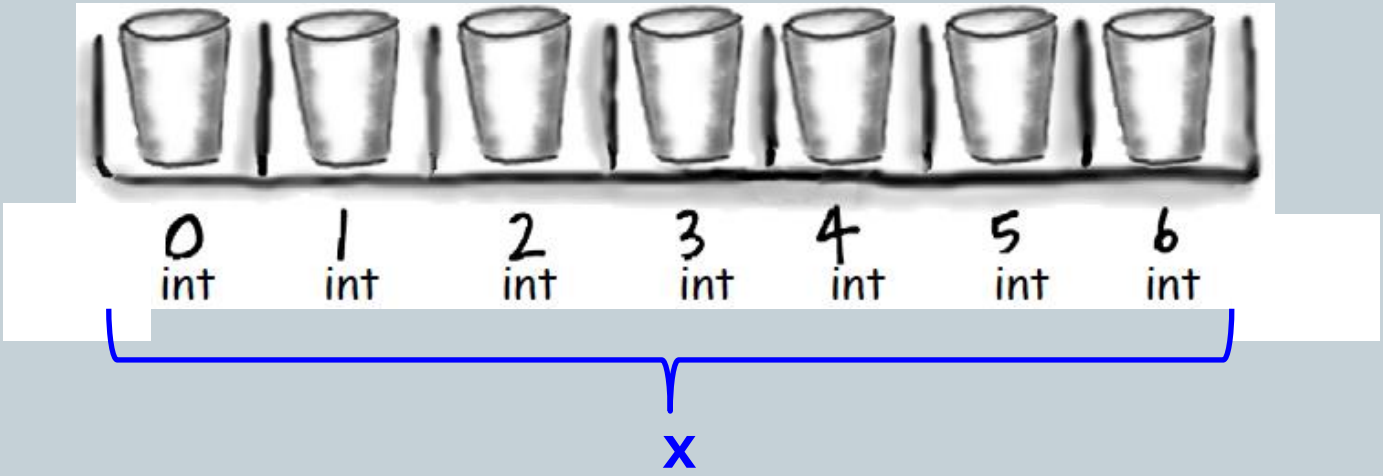
  - But x is not very useful until you "new" it:

```
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

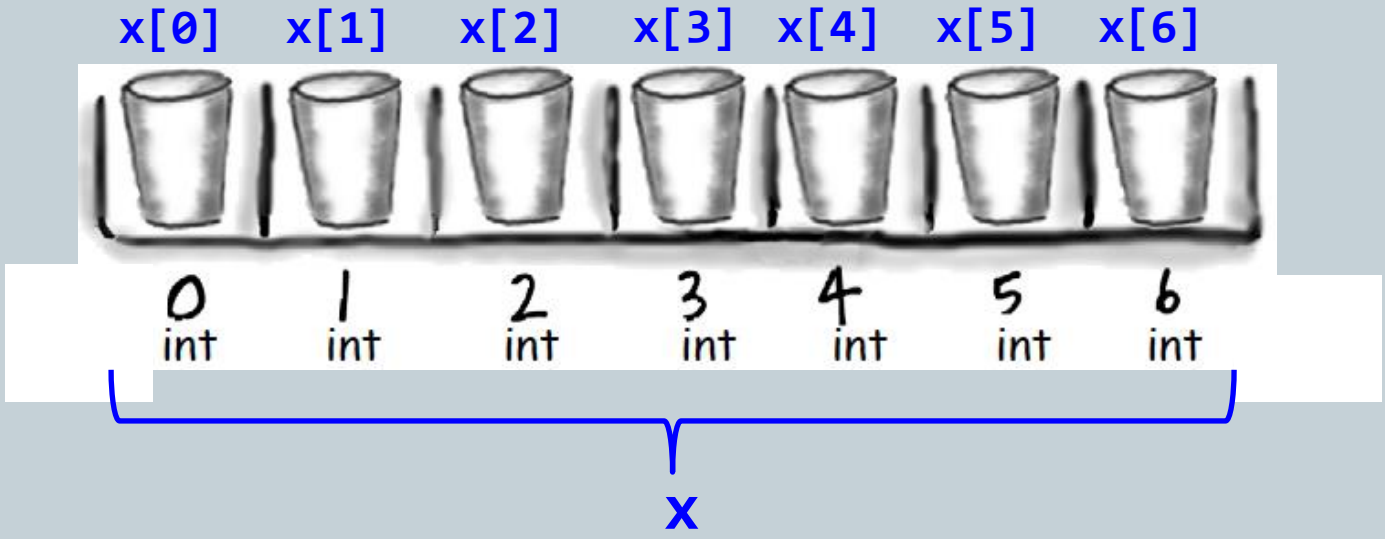


Arrays Revisited



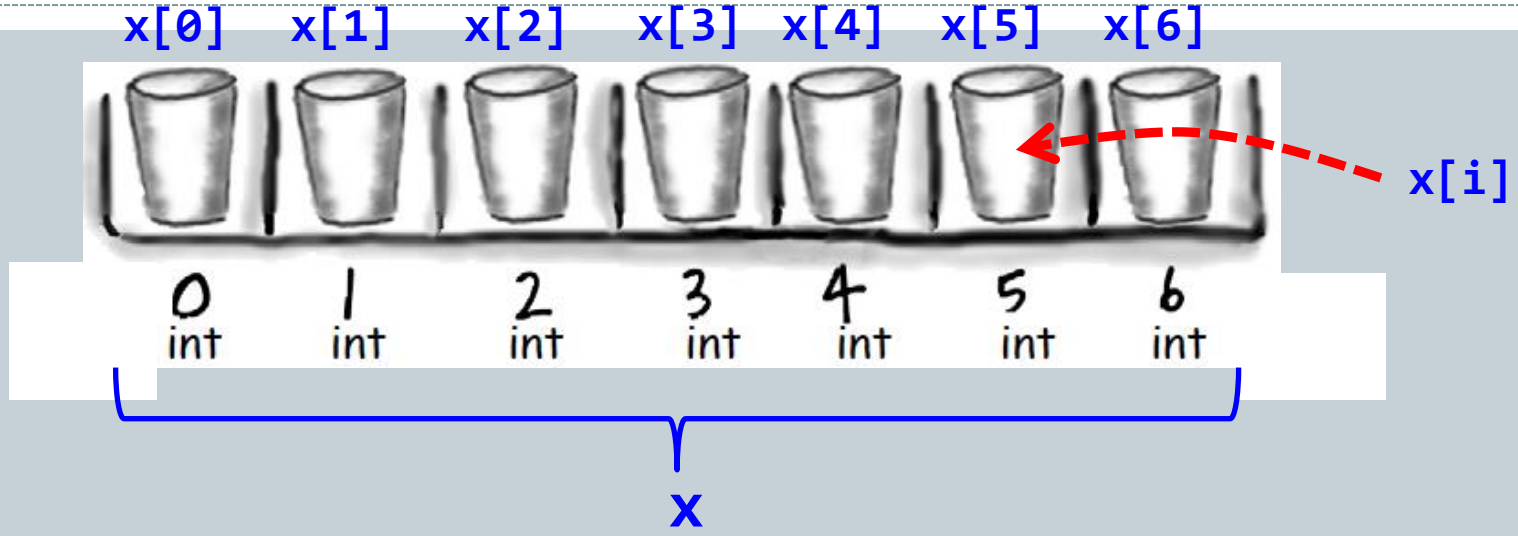
- Variable x refers to the whole set of slots
- You can't use the variable x by itself for much
- Except for finding out the number of slots: `x.length`

Arrays Revisited



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

Arrays Revisited

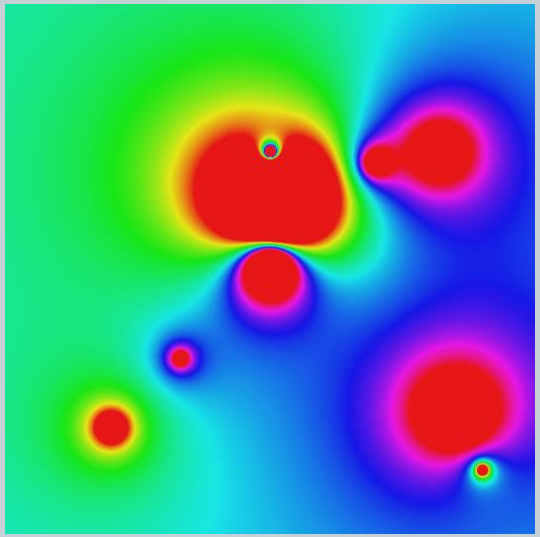


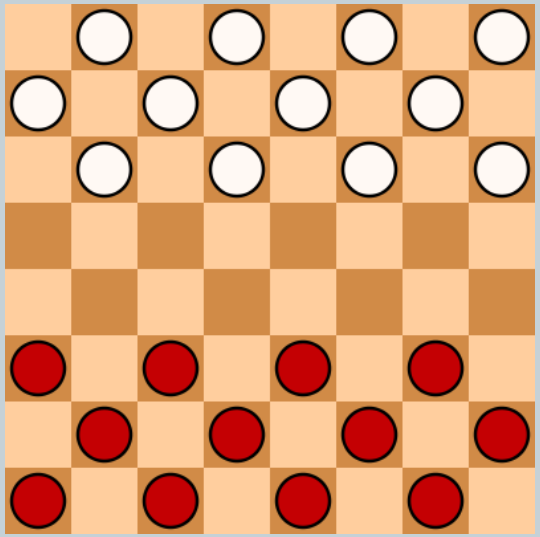
- `x[i]` refers to the value at a slot, but the slot index is determined by variable `i`
  - If i = 0 then `x[0]`, if i = 1 then `x[1]`, etc.
- Whatever inside [] must be an int
- Whatever inside [] must be in 0 to `x.length - 1` (inclusive)

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

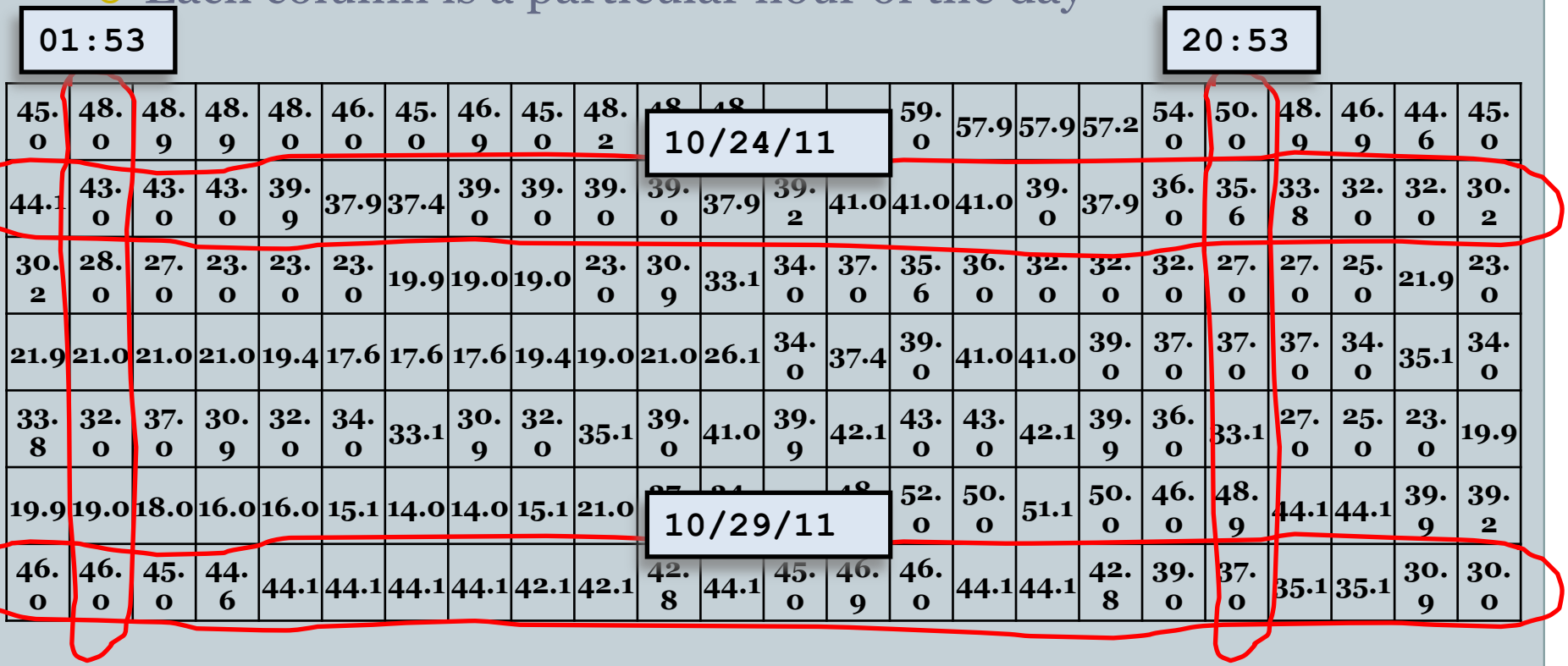
0h	1h	…	23h
32.5	30.0		45.6
…			
59.5	62.1	…	60.0
60.7	61.8	…	70.5
62.6	62.0	…	68.0





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

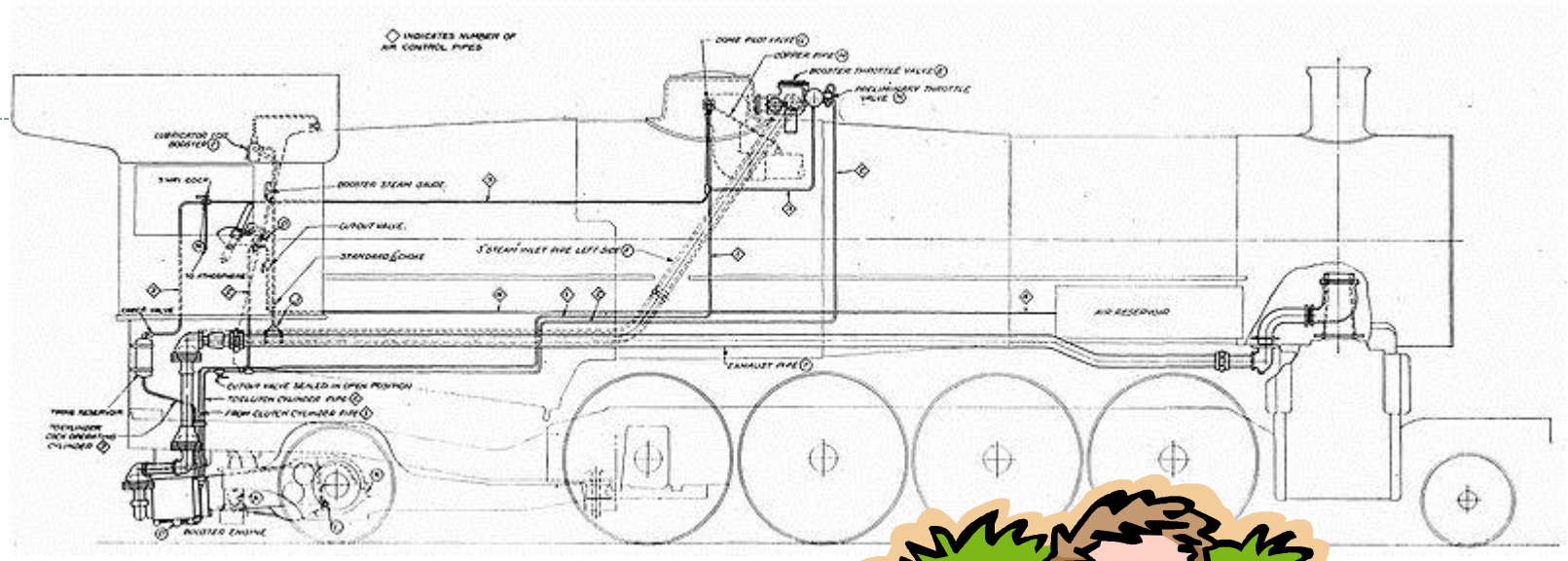
```
final int DAYS  = 7;
final int HOURS = 24;
double [][] a = new double[DAYS][HOURS];
double min = Double.POSTIVE_INFINITY;
double max = Double.NEGATIVE_INFINITY;

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);
```

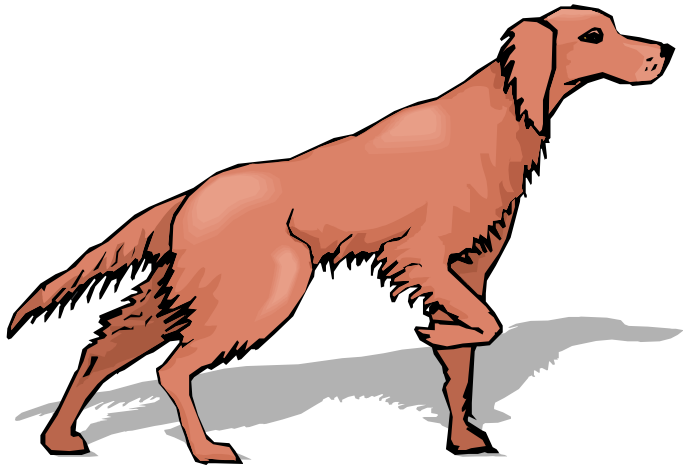
Start the min at a really high temp.

Start the max at a really low temp.

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

Object Oriented Design





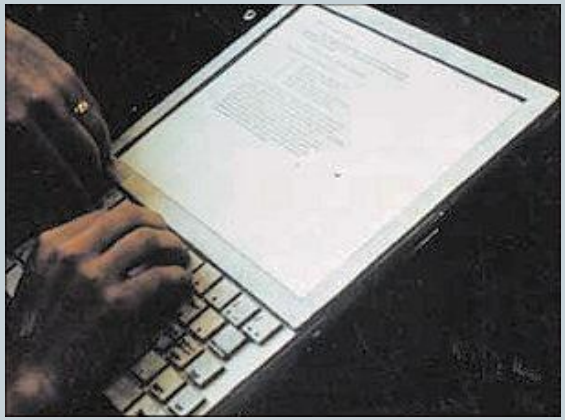
Object Oriented Programming

- Procedural programming [verb-oriented]
  - Tell the computer to do this
  - Tell the computer to do that
- OOP philosophy
  - Software simulation of real world
  - We know (approximately) how the real world works
  - Design software to model the real world
- Objected oriented programming (OOP) [noun-oriented]
  - Programming paradigm based on data types
  - Identify: objects that are part of problem domain or solution
    - Objects are distinguishable from each other (references)
  - State: objects know things (instance variables)
  - Behavior: objects do things (methods)

Alan Kay

Alan Kay
2003 Turing Award

- Alan Kay [Xerox PARC 1970s]
  - Invented Smalltalk programming language
  - Conceived portable computer
  - Ideas led to: laptop, modern GUI, OOP



Dynabook: A Personal Computer for Children of All Ages, 1968.

"The computer revolution hasn't started yet."

"The best way to predict the future is to invent it."

"If you don't fail at least 90 per cent of the time, you're not aiming high enough."

— Alan Kay

Data Encapsulation

- Data type (aka class)
  - "Set of values and operations on those values"
  - e.g. `int, String, Charge, Picture, Enemy, Player`
- Encapsulated data type
  - Hide internal representation of data type.
- Separate implementation from design specification
  - Class provides data representation & code for operations
  - Client uses data type as black box
  - API specifies contract between client and class
- Bottom line:
  - You don't need to know how a data type is implemented in order to use it

Intuition

Client

API
- volume
- change channel
- adjust picture
- decode NTSC signal

Implementation
- cathode ray tube
- electron gun
- Sony Wega 36XBR250
- 241 pounds

Client needs to know how to use API

Implementation needs to know what API to implement

Implementation and client need to agree on API ahead of time.

Intuition

Client

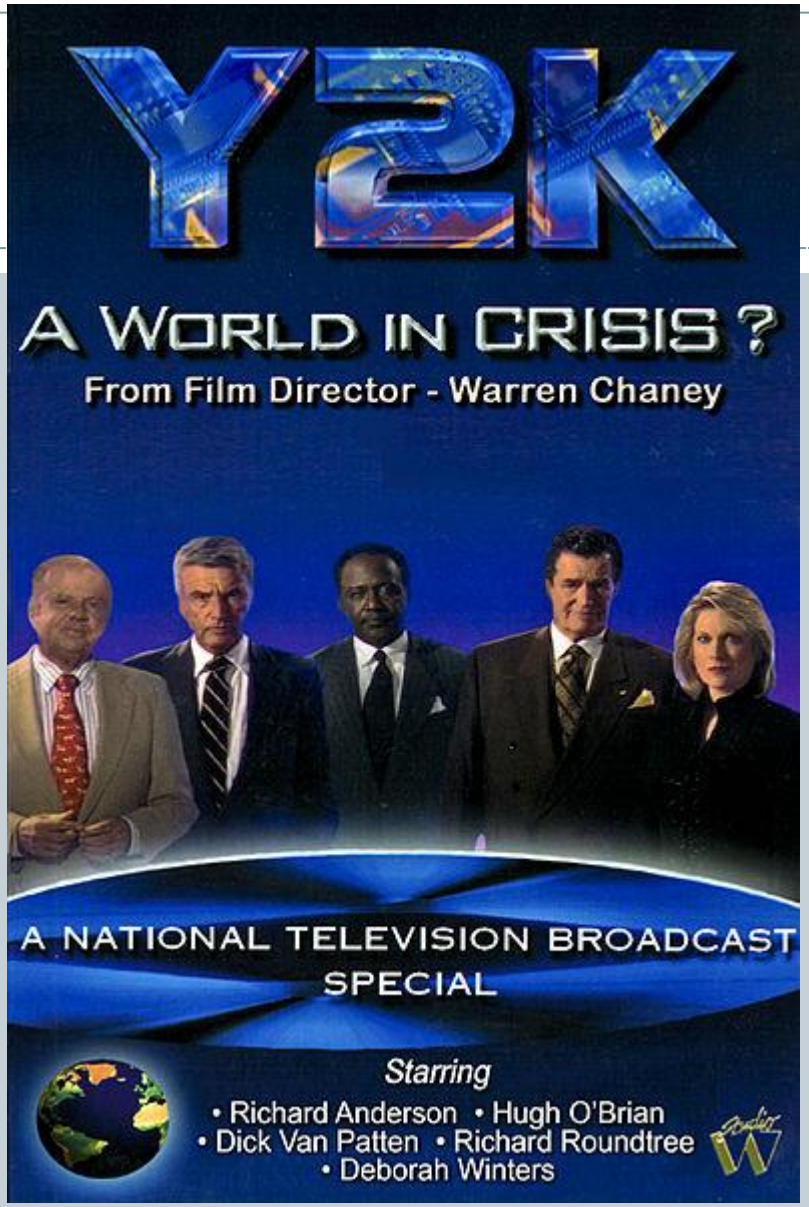
API
- volume
- change channel
- adjust picture
- decode NTSC signal

Implementation
- gas plasma monitor
- Samsung FPT-6374
- wall mountable
- 4 inches deep

Client needs to know how to use API

Implementation needs to know what API to implement

Can substitute better implementation without changing the client.



"When someone says to you, Y2K is not a problem. Inform them that it already is... one trillion dollars - and rising." --Richard Anderson

Time Bombs

- Internal representation changes
  - [Y2K] Two digit years: Jan 1, 2000
  - [Y2038] 32-bit seconds since 1970: Jan 19, 2038

- Lesson
  - By exposing data representation to client, may need to sift through millions of lines of code to update

http://xkcd.com/607/

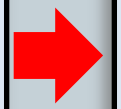
Access Modifiers

- Access modifier
  - All instance variables and methods have one:
    - `public` - everybody can see/use
    - `private` - only class can see/use
    - `default` - everybody in package (stay tuned), what you get if you don't specify an access modifier!
    - `protected` - everybody in package and subclasses (stay tuned) outside package
  - Normally:
    - Instance variables are `private`
    - API methods the world needs are `public`
    - Helper methods used only inside the class are `private`

Data Encapsulation Example

- Person class
  - Originally stored first & last name in one instance variable
  - Now we want them separated → change instance vars

```
public class Person
{
   private String name  = "";
   private double score = 0.0;

   public String toString()
   {
      return name;
   }
   ...
}
```



```
public class Person
{
   private String first = "";
   private String last  = "";
   private double score = 0.0;

   public String toString()
   {
      String result = first;
      result += " ";
      result += last;
      return result;
   }
   ...
}
```

Original version, combined names

New version, names separated.

Non-encapsulated Example

- What if instance variables were `public`?
  - Client program might use instead of methods

```
public class Person
{
   public String first = "";
   public String last  = "";
   public double score = 0.0;

   public String toString()
   {
      String result = first;
      result += " ";
      result += last;
      return result;
   }
   ...
}
```

Non-encapsulated version, instance variables are `public`.

```
...
Person p = new Person("Bob Dole");

System.out.println(p.name +
                   " " +
                   p.score);

...
```

Client program.
Changing instance variables causes compile error. Client should have been using `toString()` but used instance variable because they were publically available. Code like this might be in many client programs!

Getters and Setters

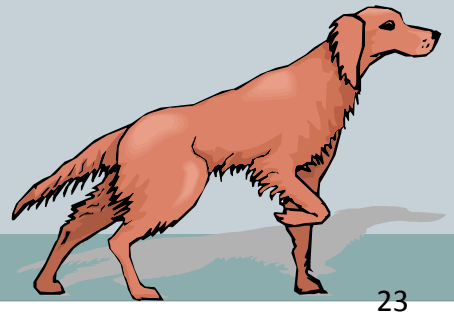
- Encapsulation does have a price
  - If clients need access to instance var, must create:
    - getter methods - "get" value of an instance var
    - setter methods - "set" value of an instance var

```
public double getPosX()
{
   return posX;
}
```

Getter method.
Also know as an ***accessor*** method.

```
public void setPosX(double x)
{
   posX = x;
}
```

Setter method.
Also know as a ***mutator*** method.



Immutability

- Immutable data type
  - Object's value cannot change once constructed

mutable	*immutable*
`Picture`	`Charge`
`Histogram`	`Color`
`Turtle`	`Stopwatch`
`StockAccount`	`Complex`
`Counter`	`String`
Java arrays	primitive types

Immutability: Pros and Cons

- Immutable data type
  - Object's value cannot change once constructed
- Advantages
  - Avoid aliasing bugs
  - Makes program easier to debug
  - Limits scope of code that can change values
  - Pass objects around without worrying about modification
- Disadvantage
  - New object must be created for every value

String Immutability: Consequences

```
String s = "Hello world!";
System.out.println("before : " + s);
s.toUpperCase();
System.out.println("after  : " + s);
```

```
before : Hello world!
after  : Hello world!
```

Since `String` is immutable, this method call *cannot* change the variable s!

```
String s = "Hello world!";
System.out.println("before : " + s);
s = s.toUpperCase();
System.out.println("after  : " + s);
```

```
before : Hello world!
after  : HELLO WORLD!
```

Final Access Modifier

- Final
  - Declaring variable **final** means that you can assign value only once, in initializer or constructor

```
public class Counter
{
    private final String name;
    private int count;
    ...
}
```

This value doesn't change once the object is constructed

This value can change in instance methods

- Advantages
  - Helps enforce immutability
  - Prevents accidental changes
  - Makes program easier to debug
  - Documents that the value cannot not change

Equality: Integer Primitives

- Boolean operator ==
  - See if two variables are exactly equal
    - i.e. they have identical bit patterns
- Boolean operator !=
  - See if two variables are NOT equal
    - i.e. they have different bit patterns

```
int a = 5;

if (a == 5)
   System.out.println("yep it's 5!");

while (a != 0)
  a--;
```

This is a safe comparison since we are using an integer type.

Equality: Floating-point Primitives

- Floating-point primitives
  - i.e. `double` and `float`
  - Only an approximation of the number
  - Use == and != at your own peril

```
double a = 0.1 + 0.1 + 0.1;
double b = 0.1 + 0.1;
double c = 0.0;

if (a == 0.3)
   System.out.println("a is 0.3!");

if (b == 0.2)
   System.out.println("b is 0.2!");

if (c == 0.0)
   System.out.println("c is 0.0!");
```

```
b is 0.2!
c is 0.0!
```

Equality: Floating-point Primitives

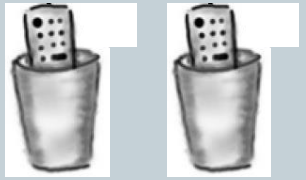
- Floating-point primitives
  - i.e. `double` and `float`
  - Only an approximation of the number
  - Use == and != at your own peril

```
double a = 0.1 + 0.1 + 0.1;
double b = 0.1 + 0.1;
double c = 0.0;
final double EPSILON = 1e-9;

if (Math.abs(a - 0.3) < EPSILON)
   System.out.println("a is 0.3!");

if (Math.abs(b - 0.2) < EPSILON)
   System.out.println("b is 0.2!");

if (Math.abs(c) < EPSILON)
   System.out.println("c is 0.0!");
```

```
a is 0.3!
b is 0.2!
c is 0.0!
```

Equality: Reference Variables

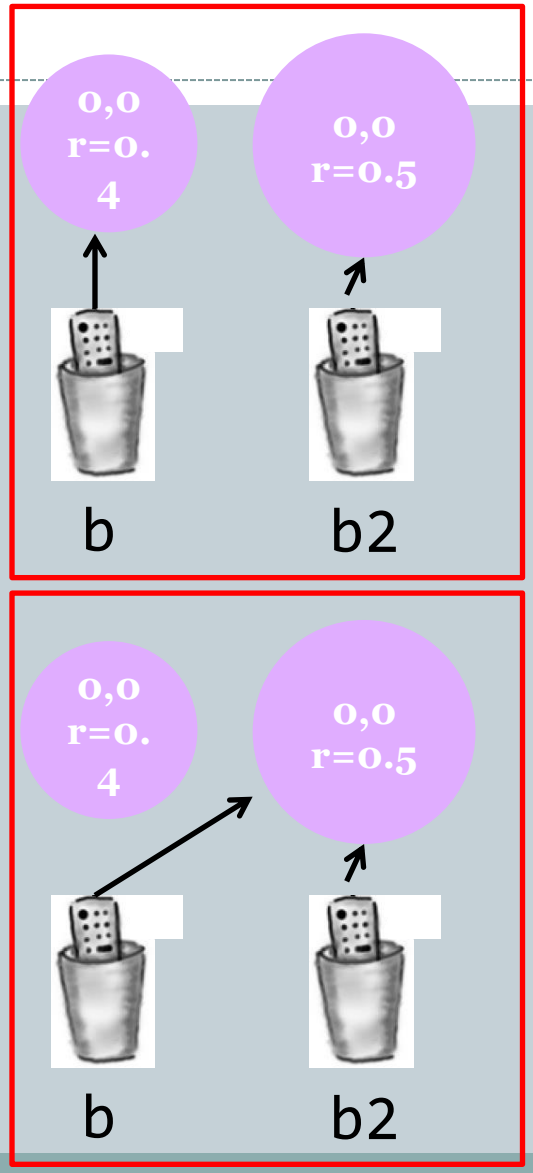
- Boolean operator ==, !=
  - Compares bit values of remote control
    - Not the values stored in object's instance variables
  - Usually not what you want



```
Ball b = new Ball(0.0, 0.0, 0.5);
Ball b2 = new Ball(0.0, 0.0, 0.5);

if (b == b2)
   System.out.println("balls equal!");

b = b2;
if (b == b2)
   System.out.println("balls now equal!");
```

Equality: Reference Variables

```
Ball b = new Ball(0.0, 0.0, 0.4);
Ball b2 = new Ball(0.0, 0.0, 0.5);



if (b == b2)
   System.out.println("balls equal!");




b = b2;
if (b == b2)
   System.out.println("balls now equal!");
```




```
balls now equal
```

Object Equality

- Implement `equals()` instance method
  - Up to class designer exactly how it works
  - Client needs to call `equals()`, not == or !=

```
public class Ball
{
   // See if this Ball is at the same location and radius
   // as some other Ball (within a tolerance of 1e-10).
   // Ignores the color.
   public boolean equals(Ball other)
   {
      final double EPSILON = 1e-9;
      return ((Math.abs(posX   - other.posX)   < EPSILON) &&
              (Math.abs(posY   - other.posY)   < EPSILON) &&
              (Math.abs(radius - other.radius) < EPSILON));
   }
   ...
}
```

Equality: String Variables

- Boolean operator ==, !=
  - Compares bit values of remote control
    - A `String` is a reference variable
    - Does *not* compare text stored in the `String` objects
  - Usually *not* what you want

```
String a = "hello";
String b = "hello";
String c = "hell" + "o";
String d = "hell";
d = d + "o";

if (a == b) System.out.println("a equals b!");
if (b == c) System.out.println("b equals c!");
if (c == d) System.out.println("c equals d!");
```

```
a equals b!
b equals c!
```

Handy String Methods

Method	
`int length()`	How many characters in this string
`char charAt(int index)`	`char` value at specified index
`String substring(int start, int end)`	Substring [`start, end - 1`] inclusive
`boolean equals(String other)`	Is this string the same as another?
`boolean equalsIgnoreCase(String other)`	Is this string the same as another ignoring case?
`String trim()`	Remove whitespace from start/end
`String toLowerCase()`	Return new string in all lowercase
`String toUpperCase()`	Return new string in all uppercase
`int indexOf(String str)`	Index of first occurrence of specified substring, -1 if not found
`int indexOf(String str, int from)`	Index of next occurrence of substring starting from index `from`, -1 if not found

Equality: String Variables

- Check equality with `equals()` method
  - Each letter must be the same (including case)

```
String a = "hello";
String b = "hello";
String c = "hell" + "o";
String d = "hell";
d = d + "o";

if (a.equals(b)) System.out.println("a equals b!");
if (b.equals(c)) System.out.println("b equals c!");
if (c.equals(d)) System.out.println("c equals d!");
```

```
a equals b!
b equals c!
c equals d!
```

Summary

- Arrays Revisited
  - Multidimensional arrays
- Object-Oriented Design
  - Data encapsulation
  - Checking for equality