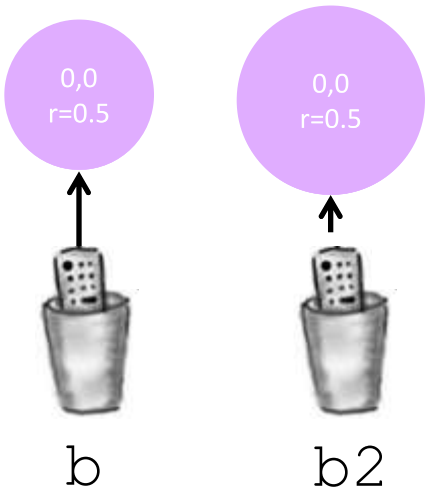
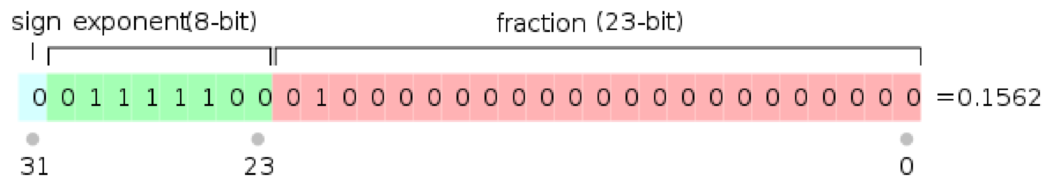
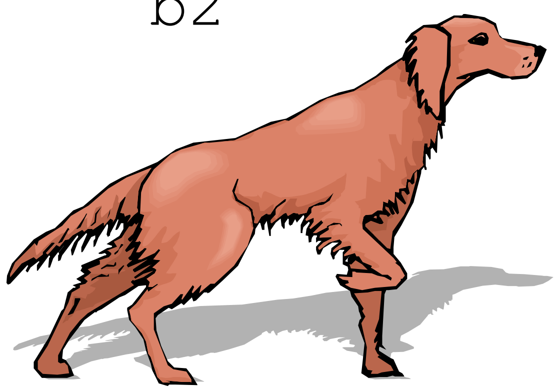
# Equality, access modifiers, strings



blah blah blah blah blah blah blah blah blah blah blah blah blah
blah blah blah blah blah blah blah blah blah blah blah blah blah
blah blah blah blah blah blah blah blah blah blah blah blah blah
blah blah blah blah blah blah blah blah blah blah blah blah blah
blah blah blah blah blah blah blah blah blah blah blah blah blah
blah blah blah blah blah blah blah blah blah blah blah blah blah
blah blah blah blah blah blah blah blah blah blah blah blah blah

# Overview

- Equality and inequality
  - Not always that simple:
    - floating-point variables
    - reference variables
- Data encapsulation
  - Important consideration when designing a class
  - Access modifiers
- Changing parameter values
  - Primitive types, reference types, arrays, `String`
- `String` class
  - Methods
  - Efficiency

# 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.

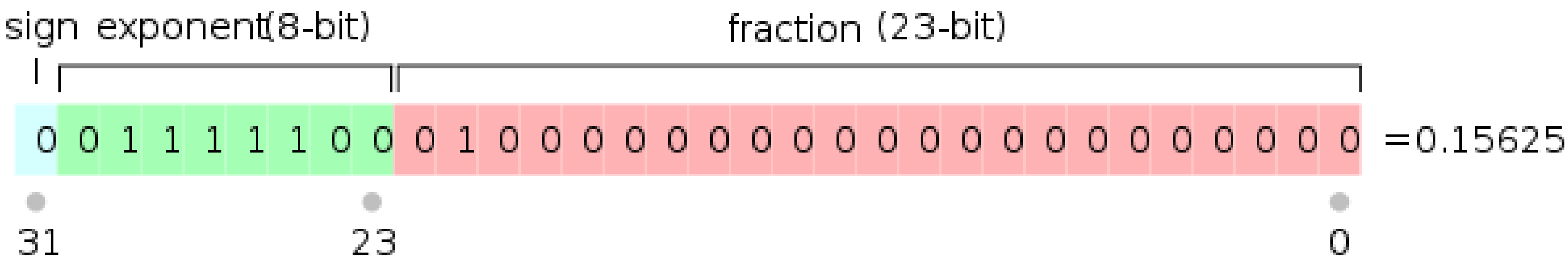
# Equality: floating-point primitives

- 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!");
```

# Equality: floating-point primitives

- Floating-point primitives
  - i.e. `double` and `float`
  - Only an approximation of the number




Floating-point numbers are shoved into a binary number.
32 bits for a `float`, 64-bits for a `double`

# 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!");
```

This works as long as no calculation has been done on 0.0 value and both are typed `double`.

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

# Safe floating-point equality check

- Floating-point primitives
  - Check if sufficiently close to target value

```
double a = 0.1 + 0.1 + 0.1;
double b = 0.1 + 0.1;
double c = 0.0;
final double EPSILON = 1e-10;

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

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

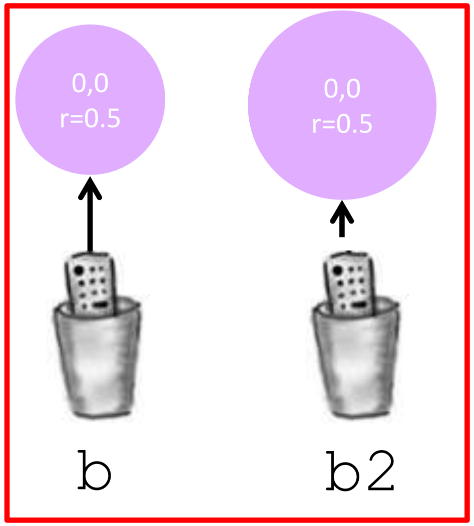
# Equality: reference variables

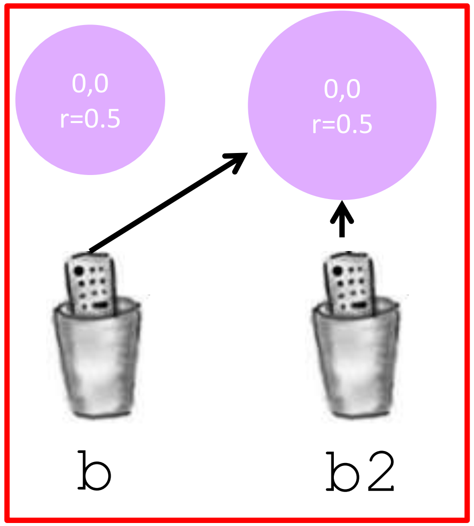
- Boolean operator ==, !=
  - Compares bit values of remote control
    - Not the values stored in object's instance variables
  - Usually not what you want

```
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

```java
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!");
```




```
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-10;
      return ((Math.abs(posX   - other.posX)   < EPSILON) &&
              (Math.abs(posY   - other.posY)   < EPSILON) &&
              (Math.abs(radius - other.radius) < EPSILON));
   }
   ...
}
```

# Access modifiers

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

# Data encapsulation

- Data encapsulation
  - Hides implementation details of an object
  - Clients don't have to care about details
  - Allows class designer to change implementation
    - Won't break previously developed clients
  - Don't expose implementation details
    - Use private access modifier

# Data encapsulation example

- Person class
  - Originally stored first and last name together in one instance variable
  - Now we want them separated
  - Change instance variables

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

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

Original version, combined names

```
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;
   }
   ...
}
```

New version, names separated.

# Non-encapsulated example

- 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 it was publically available.

# Getters and setters

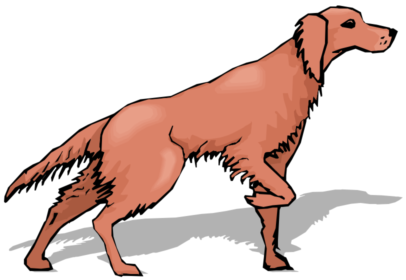
- Encapsulation does have a price
  - If clients need access to instance var, must create:
    - getter methods - "get" value of an instance var
    - setter methods - "set" value of an instance var

```
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.



# Pass by value: primitive types

- Java passes parameters by value (by copy)
  - Changes to primitives do not persist after method
    - Applies to instance methods as well as static methods
    - Primitive types: `int, double, char, long, boolean, byte, short, float`

```
public void move(double deltaX, double deltaY)
{
   posX += deltaX;
   posY += deltaY;
   deltaX = 0.0;
   deltaY = 0.0;
}
```

This has no effect on the values of `mx` and `my` in the client program.

```
Ball big = new Ball(0.0, 0.0, 0.1);
double mx = 0.5;
double my = -0.3;
big.move(mx, my);
System.out.println("mx=" + mx + " my=" + my);
```

```
mx=0.5 my=-0.3
```

# Changing elements of an array parameter

- Changes to reference types CAN persist
  - References types: arrays, objects

This changes the elements of the passed in array.

The changes persist after the method call!

```
public void move(double [] vals)
{
   posX += vals[0];
   posY += vals[1];
   vals[0] = 0.0;
   vals[1] = 0.0;
}
```

```
d0=0.0 d1=0.0
```

```
Ball big = new Ball(0.0, 0.0, 0.1);
double [] delta = new double[2];
delta[0] = 0.5;
delta[1] = -0.3;
big.move(delta);
System.out.println("d0=" + delta[0] + " " +
                   "d1=" + delta[1]);
```

# Changing an object parameter

- Passing object parameters
  - Can be changed using methods or instance vars

This will move the passed in `Ball` object's location and change its color to black.

```
public boolean overlap(Ball other)
{
    double deltaX = posX - other.posX;
    double deltaY = posY - other.posY;
    double d = Math.sqrt(deltaX * deltaX +
                         deltaY * deltaY);

    other.move(100.0, 100.0);
    other.color = new Color(0.0f, 0.0f, 0.0f);

    return (d < (radius + other.radius));
}
```

# `String` parameters

- String is a reference type
  - But it is a special kind called immutable
  - Changes to `String` parameters do NOT persist

```
// Change the name of this Person object
public void changeName(String newName)
{
   name = newName;
   newName = "";
}
```

```
first=bob
```

```
Person me = new Person("keith");
String first = "bob";
me.changeName(first);
System.out.println("first=" + first);
```

Since `String` is an immutable reference type, this change does NOT persist.

# new'd reference parameters

- Using new on reference parameter
  - Changes to new'd variable do NOT persist

We end up creating a local array.

The changes do not persist after the method call since changes were not actually to memory of `delta` array

```
public void move(double [] vals)
{
    posX += vals[0];
    posY += vals[1];
    vals = new double[2];
    vals[0] = 0.0;
    vals[1] = 0.0;
}
```

```
d0=0.5 d1=-0.3
```

```
Ball big = new Ball(0.0, 0.0, 0.1);
double [] delta = new double[2];
delta[0] = 0.5;
delta[1] = -0.3;
big.move(delta);
System.out.println("d0=" + delta[0] + " " +
                   "d1=" + delta[1]);
```

# Handy String methods

- `String` is an object with lots of 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 |

# String search example

- Goal: count occurrences of a string in a file

Call me Ishmael. Some years ago- never mind how long precisely- having little or no money in my purse, and nothing particular to interest me on shore, I thought I would sail about a little and see the watery part of the world. It is a way I have of driving off the spleen and regulating the circulation. Whenever I find myself growing grim about the mouth; whenever it is a damp, drizzly November in my soul; whenever I find myself involuntarily pausing before coffin warehouses, and bringing up the rear of every funeral I meet; and

mobydick.txt

```
% java StringCounter call < mobydick.txt
293
% java StringCounter Call < mobydick.txt
293
% java StringCounter "call me" < mobydick.txt
3
% java StringCounter Keith < mobydick.txt
0
```

# String counter program

```
public static void main(String [] args)
{
   String find = args[0];
   String text = StdIn.readAll();
   text = text.toLowerCase();
   find = find.toLowerCase();

   int index = 0;
   int count = 0;
   while (index != -1)
   {
      index = text.indexOf(find, index);
      if (index != -1)
      {
          count++;
          index = index + find.length();
      }
    }
    System.out.println(count);
}
```

# String efficiency

- Normal `String` class is immutable
  - Every time you append:
    - New object created, old one destroyed
    - This can really slow things down

```
public class StringEfficiency
{
    public static void main(String [] args)
    {
        String str   = "";
        int    num   = Integer.parseInt(args[0]);
        long   start = System.currentTimeMillis();

        for (int i = 0; i < num; i++)
            str = str + "blah ";

        long elapsed = System.currentTimeMillis() - start;
        System.out.println("Time = " + (elapsed / 1000.0));
    }
}
```

# String efficiency

```
public class StringTest
{
    public static void main(String [] args)
    {
        String str   = "";
        int    num   = Integer.parseInt(args[0]);
        long   start = System.currentTimeMillis();

        for (int i = 0; i < num; i++)
            str = str + "blah ";

        long elapsed = System.currentTimeMillis() - start;
        System.out.println("Time = " + (elapsed / 1000.0));
    }
}
```

```
% java StringTest 10000
Time = 0.383
```

```
% java StringTest 20000
Time = 1.358
```

```
% java StringTest 40000
Time = 5.551
```

```
% java StringTest 80000
Time = 25.935
```

```
% java StringTest 160000
Time = 112.63
```

```
% java StringTest 320000
Time = 477.454
```

# StringBuilder

```
public class StringTest2
{
    public static void main(String [] args)
    {
        StringBuilder str = new StringBuilder();
        int    num   = Integer.parseInt(args[0]);
        long   start = System.currentTimeMillis();

        for (int i = 0; i < num; i++)
            str.append("blah ");

        long elapsed = System.currentTimeMillis() - start;
        System.out.println("Time = " + (elapsed / 1000.0));
    }
}
```

```
% java StringTest2 10000
Time = 0.0040
```

```
% java StringTest2 20000
Time = 0.0070
```

```
% java StringTest2 40000
Time = 0.0090
```

```
% java StringTest2 80000
Time = 0.011
```

```
% java StringTest2 160000
Time = 0.015
```

```
% java StringTest2 320000
Time = 0.019
```

# Summary

- Equality
  - Usually avoid == or != with floating-point types
  - Usually avoid == or != with reference types
    - Implement an `equals()` method
- Passing parameters
  - Changing primitive → does NOT persist
  - Changing element in array → persists
  - Changing an object (via method/instance var) → persists
- Strings
  - Small stuff, use `String`
  - Big strings, use `StringBuilder`