

---

# CS11 – Java

---

Fall 2009-2010

Lecture 2

---

# Java and Object-Oriented Programming

- Java is an object-oriented programming (OOP) language.
    - In Java, programs are entirely composed of classes and objects
  - OOP is primarily about program structure
    - Doesn't have so much to do with the computational model
-

---

# Terminology: Classes and Objects

- Objects are a tight pairing of two things:
    - State – a number of related data values
    - Behavior – code that acts on those data values in coherent ways
    - “Objects = Code + Data”
  - A class is a “blueprint” for objects
    - The class defines the state and behavior of objects of that class
    - Actually defines a new type in the language
-

---

# Terminology: Fields and Methods

- A class is comprised of members
  - Fields are variables associated with the class.
    - They store the class' state.
  - Methods are operations that the class can perform.
    - A class' set of methods specifies its behavior
    - The actual code for a method is its implementation
    - These methods generally (but not always) involve the class' fields as well
-

---

# Special Methods

- Constructors create new instances of a class.
    - Can take arguments, but not required. No return value.
    - All classes have at least one constructor.
  - Accessors allow internal data to be retrieved.
    - Provides control over how data is exposed.
  - Mutators allow internal data to be modified.
    - Provides control over how and when changes can be made.
  - No destructors in Java!
  - Not all classes have accessors and mutators.
-

---

# Class Design Example

- Write a class to control a digital camera
  - Fields: shutter speed, aperture setting, zoom, focus
  - Methods: determine exposure settings, take photo
  - Control of camera requires *many* computations and low-level interactions with hardware
  - Should other components have to deal with these complexities?
  - Are all desired settings valid?
    - Limit settings to take properly exposed photos
    - Disallow configurations limited by hardware constraints
-

---

# Who Uses A Class?

- The designer or developer of the class
    - Decides how to implement the class
    - Also decides what functionality the class exposes, and what the class ought to hide.
  - The user or client of the class
    - Doesn't care so much about its implementation.
    - Just wants to use it to do some work!
  - Sometimes these are the same person, but *often* this is not the case.
-

# Abstraction and Encapsulation



## ■ Abstraction:

- Present a clean, simplified interface
- Hide unnecessary detail from users of the class (e.g. implementation details)
  - They usually don't care about these details!
  - Let them concentrate on the problem they are solving.

## ■ Encapsulation:

- Allow an object to protect its internal state from external access and modification
- The object itself governs all internal state-changes
  - Methods can ensure only valid state changes

---

# Access-Modifiers

- Can be used on classes, methods and fields
  - Four access modifiers in Java
    - **public** – Anybody can access it
    - **private** – Only the class itself can access it
    - **protected** – We'll get to this later...
    - Default access-level (if you don't specify anything)
      - Called "package-private" access
  - Protect implementation details by using access modifiers in your code!
-

```
public class Point2d {
    // Coordinates
    private double xCoord;
    private double yCoord;

    /** Two-argument constructor. */
    public Point2d(double x, double y) {
        xCoord = x;
        yCoord = y;
    }

    /** Default constructor; initializes to (0, 0). */
    public Point2d() {
        // Call 2-argument constructor
        this(0, 0);
    }

    public double getX() { return xCoord; } // Accessors
    public double getY() { return yCoord; }

    public void setX(double x) { xCoord = x; } // Mutators
    public void setY(double y) { yCoord = y; }
}
```

---

# Java Method Naming Conventions

- Java accessors usually start with **get**
    - `double getX()`
    - `double getY()`
  - Java mutators usually start with **set**
    - `void setX(double)`
    - `void setY(double)`
  - Accessors that return **boolean** often start with **is**
    - `boolean isRunning()`
    - `boolean isLoaded()`
    - Exceptions are allowed when “is” doesn’t make sense:
      - `boolean contains(Object)`
      - `boolean intersects(Set)`
-

---

# Using the Point

- Create a new `Point2d` object using the `new` operator

```
Point2d p1 = new Point2d();
```

```
Point2d p2 = new Point2d(3.04, -5.612);
```

- Call methods on the `Point2d` objects

```
p1.setX(15.1);
```

```
p1.setY(12.67);
```

```
System.out.println("p2 = (" + p2.getX() +  
    ", " + p2.getY() + ")");
```

---

---

# Objects and References

- What are **p1** and **p2** ?

```
Point2d p1 = new Point2d();
```

```
Point2d p2 = new Point2d(3.04, -5.612);
```

- They are references to **Point2d** objects
- They are *not* objects themselves

- Juggling references:

```
Point2d p3 = p1; // Still only two objects
```

```
p1 = null; // Both objects still reachable
```

```
p2 = null; // One object isn't reachable!
```

- JVM tracks when objects are no longer reachable
    - “Garbage collection”
-

---

# Object Method-Arguments in Java

- What happens when you call a function with an object argument?

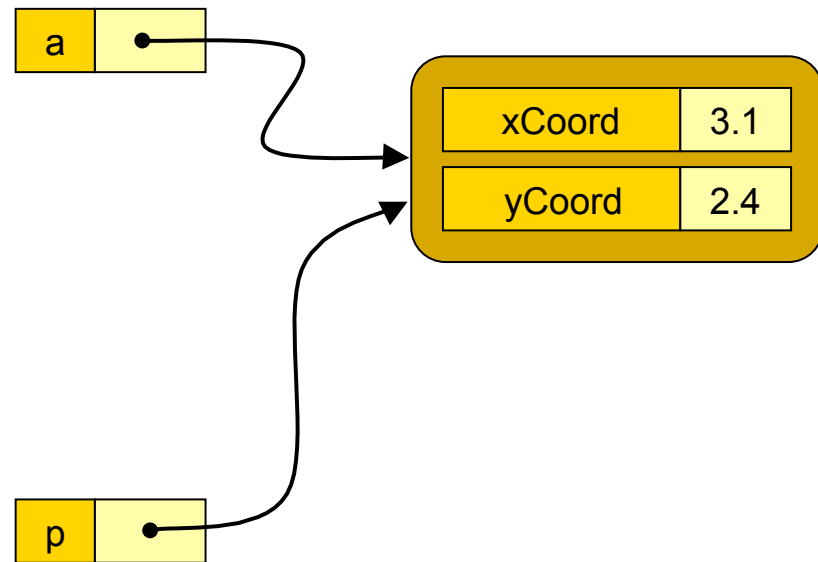
```
public void printPoint(Point2d p)
```

- Remember, `p` is a reference to the object
  - Reference is copied into `p`, but the `Point2d` object that it refers to is *not*
  - Side-effects and funky bugs can easily occur!
-

# Passing Objects in Java

```
void main(String[] args) {  
    Point2d a =  
        new Point2d(3.1, 2.4);  
  
    printPoint(a);  
}
```

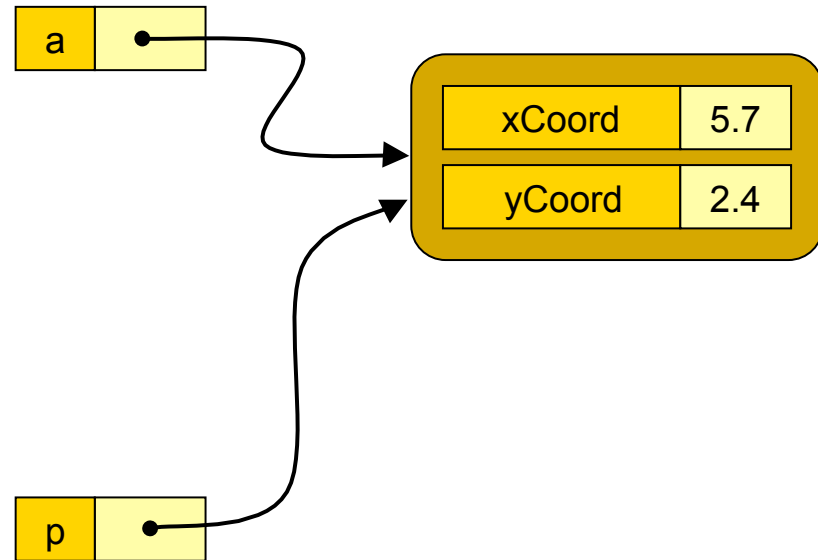
```
void printPoint(Point2d p) {  
    System.out.println(p.getX() +  
        ", " + p.getY());  
    p.setX(5.7); // ???  
}
```



# Passing Objects in Java (2)

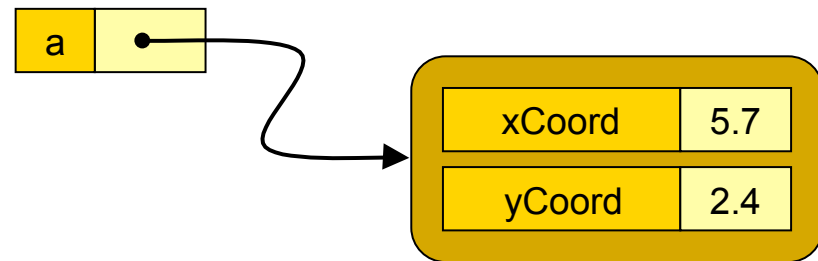
```
void main(String[] args) {  
    Point2d a =  
        new Point2d(3.1, 2.4);  
  
    printPoint(a);  
}
```

```
void printPoint(Point2d p) {  
    System.out.println(p.getX() +  
        ", " + p.getY());  
    p.setX(5.7); // affects a  
    p = new Point2d(-6.9, 0.7); // ???
```

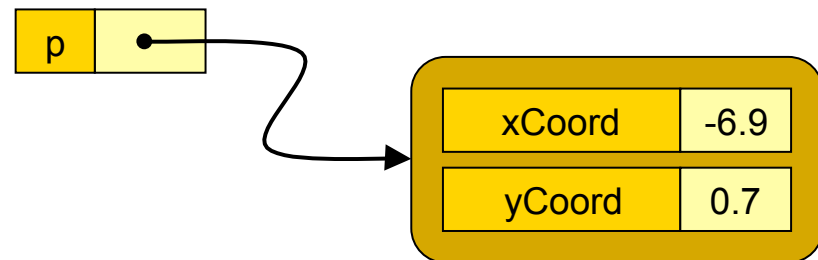


# Passing Objects in Java (3)

```
void main(String[] args) {  
    Point2d a =  
        new Point2d(3.1, 2.4);  
  
    printPoint(a);  
}
```

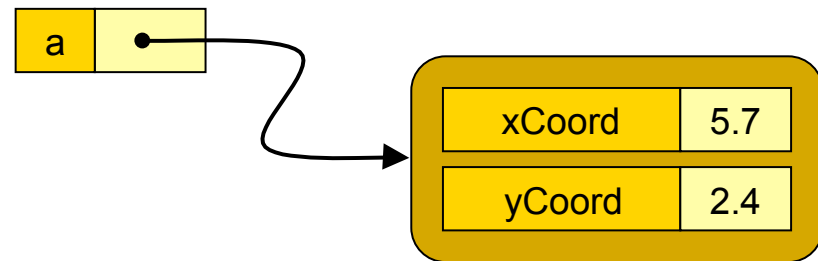


```
void printPoint(Point2d p) {  
    System.out.println(p.getX() +  
        ", " + p.getY());  
    p.setX(5.7); // affects a  
    p = new Point2d(-6.9, 0.7);  
    p.setY(-2.1); // ???  
}
```

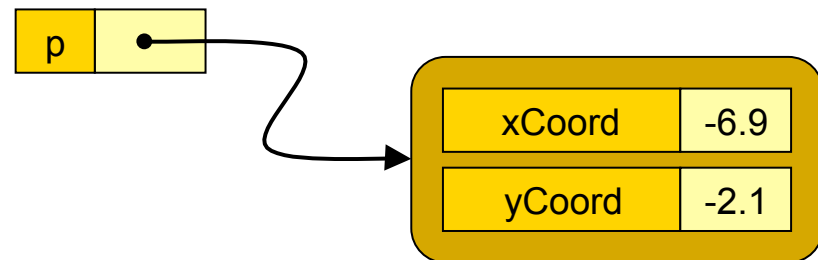


# Passing Objects in Java (4)

```
void main(String[] args) {  
    Point2d a =  
        new Point2d(3.1, 2.4);  
  
    printPoint(a);  
}
```



```
void printPoint(Point2d p) {  
    System.out.println(p.getX() +  
        ", " + p.getY());  
    p.setX(5.7); // affects a  
    p = new Point2d(-6.9, 0.7);  
    p.setY(-2.1); // local only  
}
```



---

# The Moral

- Be very careful with object-references
    - If a method accidentally changes an object, it can be very tricky to track down.
  - Where reasonable, make objects immutable
    - Java has no equivalent to C++ `const` keyword!
    - An object is immutable if it provides no mutators
      - Set object's state at construction time
      - Don't provide any way to change the state
-

---

# Method Magic

- Most methods have an *implicit* parameter **this**
  - **this** is a reference to the object being called
- Implicitly used when object fields or methods are accessed inside another method

```
public double getX() {  
    return xCoord; // Same as "return this.xCoord;"  
}
```

```
public String toString() {  
    // Same as "this.getX()" and "this.getY()"  
    return "(" + getX() + " " + getY() + ")";  
}
```

---

---

# Method Magic (2)

- Can also use **this** to resolve ambiguities

```
void setX(double xCoord) {  
    // xCoord is the parameter  
    // this.xCoord is the object's field  
    this.xCoord = xCoord;  
}
```

- Avoid ambiguities like this when possible!  
They lead to bugs.
    - Name arguments so they don't overlap with names of fields or methods.
-

---

# Static Methods

- Some methods do not require a specific object
    - Called “static methods,” or “class methods.”

```
public static double atan2(double y, double x);
```
    - Static methods can't use **this** reference
      - Method isn't called on a specific object!
    - Specify **ClassName.methodName()**
  - Non-static methods called “instance methods”
  - **java.lang.Math** has *only* static methods

```
double tangent = Math.atan2(yComp, xComp);
```
-

---

# Equality in Java

- Primitive types use `==` the way you would expect.
- For reference types, `==` compares the references themselves!

```
Point2d p1 = new Point2d(3, 5);
```

```
Point2d p2 = new Point2d(3, 5);
```

```
Point2d p3 = p1;
```

- Points `p1` and `p3` are the *same object*
    - `p1 == p3` is **true**
    - `p1 == p2` is **false**, even though values are the same
  - Use `obj1.equals(obj2)` to test value-equality
    - Corollary: When you write classes, provide a reasonable implementation of the `equals()` method.
-

---

# The `equals ()` Method

- Signature:

```
public boolean equals(Object obj)
```

- Returns true if `obj` is “equal to” this object

- Depends on what your class represents!
- If `obj` is `null`, the answer is always “not equal”

- Note that `obj` is a generic `Object` reference

- It could be any reference-type! Check that too.
  - The `instanceof` keyword lets you do this
-

---

# Does `equals ()` Make Sense?

- Reflexive:
    - `a.equals (a)` should return true
  - Symmetric:
    - `a.equals (b)` should be the same as `b.equals (a)`
    - This can be tricky sometimes...
  - Transitive:
    - If `a.equals (b)` is true and `b.equals (c)` is true, then `a.equals (c)` should also be true
  - Nulls:
    - `a.equals (null)` should be false
-

---

# Are These Points Equal?

```
public boolean equals(Object obj) {
    // Is obj a Point2d?
    if (obj instanceof Point2d) {
        // Cast other object to Point2d type,
        // then compare.
        Point2d other = (Point2d) obj;
        if (xCoord == other.getX() &&
            yCoord == other.getY()) {
            return true;
        }
    }

    // If we got here then they're not equal.
    return false;
}
```

---

---

# The **instanceof** Operator

- Use this to test an object's type – its class
- Defined to return false if the reference is null
  - This is why we don't need to check if the incoming object-reference is null.



---

# Why `equals` (Object) ?

- Classes can derive from other classes
    - Child class inherits all fields/methods of the parent class
    - Allows hierarchies of classes to be defined
    - Child class can be treated as its parent, since it has (at least) the same members as the parent class
  - In Java, *all* classes derive from `java.lang.Object`
    - All objects can be treated as an instance of `Object`
    - `java.lang.Object` defines functionality that *all* Java classes should provide
      - `equals()`, `hashCode()`, `getClass()`, `clone()`, etc.
    - Example: can use `equals()` to compare *any* two objects
-

---

# Class Inheritance

- More details about class inheritance in the upcoming weeks!



---

# Some More Math Details

- Types are converted and promoted as needed
    - Example: `double twoPi = 2 * 3.14159;`
  - Java complains when precision may be lost
    - Example: `float result = twoPi / 6;`
      - Compiler error: `possible loss of precision`
      - Problem: Trying to store a double-precision value into a single-precision variable.
    - Fix with explicit cast:
      - `float result = (float) (twoPi / 6);`
    - (or you could change the type of `result`)
-

---

# Mmmm, Candy

- You want to buy some candy.
    - The first candy costs 10 cents.
    - Each subsequent candy costs 10 cents more than the previous one.
      - Second candy is 20 cents.
      - Third candy is 30 cents.
      - etc.
  - You only have a dollar to spend.
  - How many candies can you buy?
-

---

# Candy-Budget Program

- Write a program to figure it out:

```
public class CandyBudget {  
  
    public static void main(String[] args) {  
        double fundsLeft = 1.00;  
        int numCandies = 0;  
        for (double price = 0.10; price <= fundsLeft;  
            price += 0.10) {  
            numCandies++;  
            fundsLeft -= price;  
        }  
  
        System.out.println("I can buy " + numCandies);  
        System.out.println("I have $" + fundsLeft +  
            " remaining.");  
    }  
}
```

---

---

# How Many??

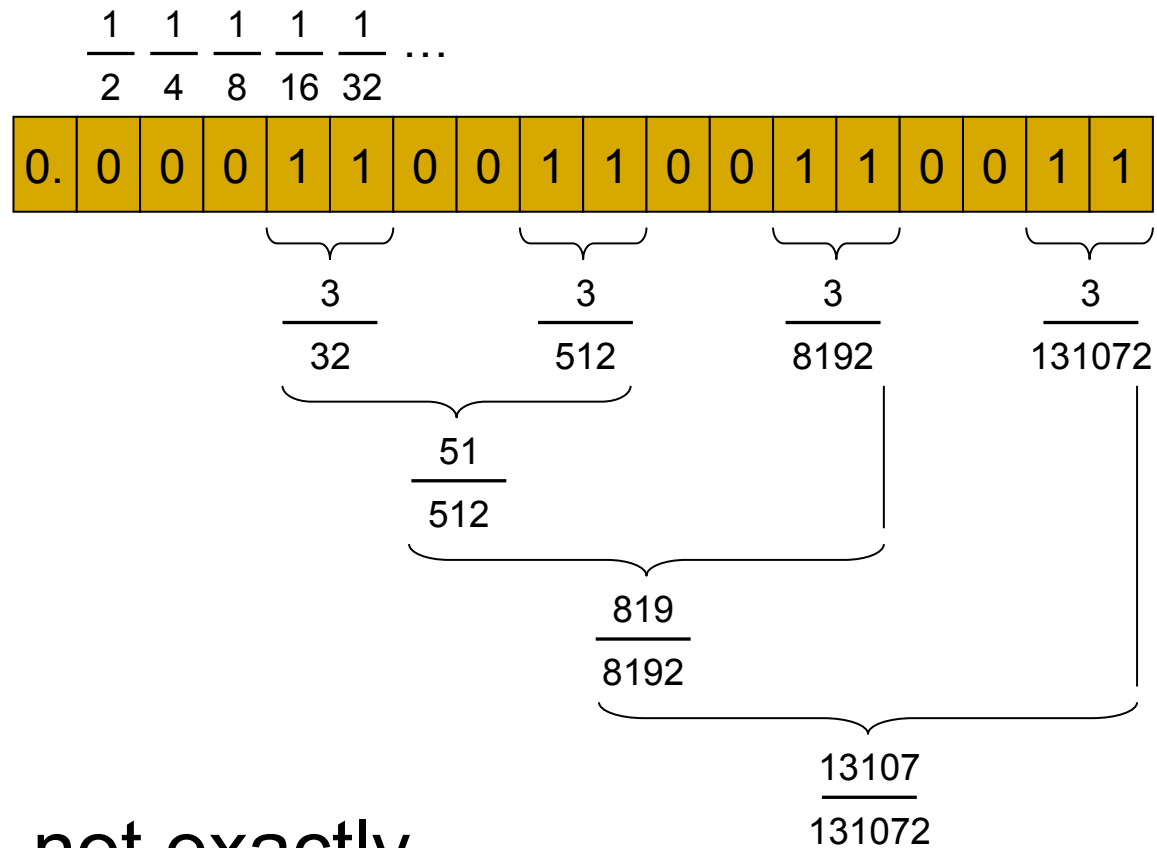
- Compile and run the program:

```
> javac CandyBudget.java
> java CandyBudget
I can buy 3 candies.
I have $0.39999999999999999 remaining.
```

- The Problem:

- Floating-point numbers are *approximate*
  - In this case, 0.1 is  $0.0\overline{0011}$  in binary
-

# Exactly What Is 0.1?



- Well, not exactly...

---

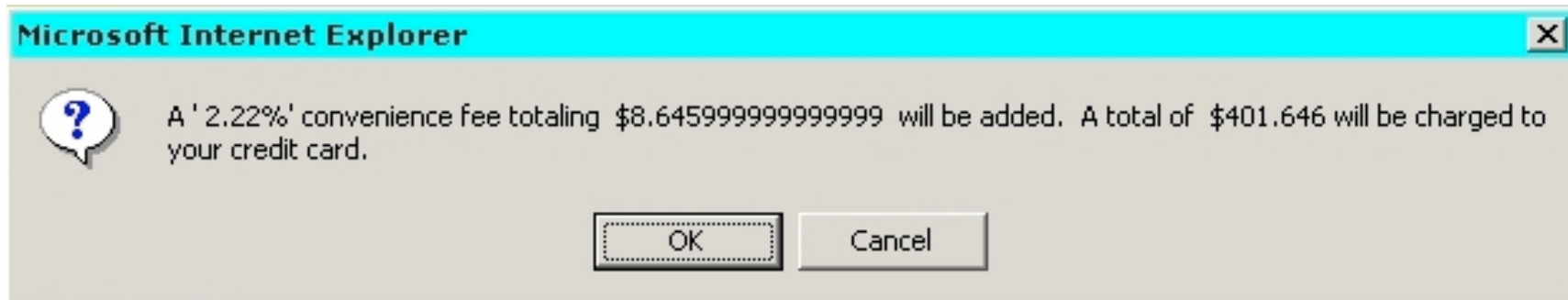
## The Moral

- Floating-point arithmetic is an *approximation*
  - Don't use it for monetary calculations
    - Use `int` or `long` and calculate in cents
    - Or, use `java.math.BigDecimal` for high-precision math
  - Floating-point math still works well in most cases
    - Be on the watch for weirdness
-

---

# The Moral (2)

- A real-world example:



---

# This Week's Homework

- Create a class to represent a 3D point!
    - Encapsulate the implementation details
    - Constructors, accessors, mutators
  - Make other useful methods for the class
    - Calculate distance to another point
    - A static method to find the area of a triangle
  - Write a program to demonstrate your class
-

---

## Next Week

- More about class hierarchies and inheritance
  - An introduction to the Swing GUI classes
-