

# CS11 – Java

---

Winter 2014-2015

Lecture 1

---

# Welcome!

- 7 Lectures
    - Slides posted on CS11 website
    - <http://courses.cms.caltech.edu/cs11>
  - 7 Lab Assignments
    - Made available around Friday
    - Due one week later – Friday, 12 noon
  - Assignments take a bit more than 3 hours per week, on average
-

# Assignments and Grading

- Labs are given a score in range 0..3, and feedback
  - If your code is broken, you will have to fix it.
  - If your code is sloppy, you will have to clean it up.
- Must receive 75% of the possible points to pass
  - With 7 assignments, 21 points available → must score 15.5 or better to pass
  - With 8 assignments, 24 points available → must score 18 or better to pass
  - Can definitely pass without completing all labs
- Please turn in assignments on time
  - You will lose 0.5 points per day on late assignments

# Lab Submissions

- Using csman homework submission website:
  - <https://csman.cms.caltech.edu>
  - Many useful features, such as email notifications
- Must have a CS cluster account to submit
  - csman authenticates against CS cluster account
- CS cluster account also great for doing labs!
  - Can easily do the labs on your own machine, since Java works the same anywhere
  - Just make sure you have Java 1.7+

---

# Course Texts

- No textbook is required
  - All necessary information is available online
    - Extensive Java API documentation
    - Java tutorials
-

---

# A Brief History of Java

- Created by Sun Microsystems starting late '90s
    - Intended for embedded-systems programming
    - Primary goal was improving on C++
    - Renamed to Java in 1994
  - Java 1.0 released in 1995
    - Versions 1.1, 1.2, 1.3. 1.4
  - Numbering scheme changed with Java 5.0
    - (SDK/development version is still called 1.5)
  - Current version is Java 8
-

# A Brief History of Java (2)

- Language, and standard libraries, have expanded dramatically over the years
  - Java 6 released in late 2006 – introduced many new language features, new APIs
  - Java 7 released in mid 2011
- Java platform was made (mostly) open-source by Sun on May 2007
  - Allows Java platform to be ported to, and customized for, additional hardware platforms
- In Jan 2010, Oracle acquired Sun
  - Caused significant concern about future of Java

# Design Goals of Java Language

- Simple and familiar
  - Based on C/C++, but with many subtleties removed
- Object-oriented
  - Well suited to distributed systems
- Architecture-neutral
  - Both source code and binaries are portable
- Dynamic loading and binding
  - Minimizes recompilations, and facilitates modularity!
- Secure
  - Class verification, code signing, permissions
- Multithreaded
  - Language specifies platform-neutral threading support



# How Java Does Its Thing

- Source code goes into `.java` files.
- One top-level class per file.
- Class' name dictates file name.
- Example: `HelloWorldApp.java`

```
// Display a message and then exit.  
public class HelloWorldApp {  
    public static void main(String[] args) {  
        System.out.println("Hello, world!");  
    }  
}
```

# How Java Does Its Thing (2)

- Java compiler takes `.java` files and compiles them into platform-independent `.class` files.

- `javac HelloWorldApp.java`

- ➔ produces `HelloWorldApp.class`

- These class files contain *byte-codes* – instructions for the Java Virtual Machine (JVM).

- Byte-codes for our example:

```
public static void main(java.lang.String[])
0: getstatic      #2;      //Field java/lang/System.out
3: ldc           #3;      //String "Hello, world!"
5: invokevirtual #4;      //Method java/io/PrintStream.println
8: return
```

# How Java Does Its Thing (3)

- Run the program with a Java Virtual Machine (JVM)
  - The JVM takes a class name, not the class' filename

```
> java HelloWorldApp
Hello, world!
```
- The **java** program implements the JVM for a specific platform
  - Can run Java on any platform with a JVM implementation. (Windows, Linux, Solaris, MacOS X, ...)
- Some JVMs improve performance by compiling Java byte-codes into native machine code
  - Called “just-in-time” compilation, or JIT for short

# Java Comments

- Java comments are just like C++ comments

```
/*  
 * This method prints hello world.  
 */  
public static void main(String[] args) {  
    // This next part is tricky...  
    System.out.println("Hello, world!"); // phew!  
}
```

- Block comments can span multiple lines
  - Single-line comments extend to end of line
- Use block comments before classes/functions
- Use single-line comments within functions

# Java Data Types

## ■ Primitive Types

- ❑ **boolean**      values are **true** or **false**
- ❑ **char**          16-bit unsigned integer  
                         (for Unicode characters)
- ❑ **byte**            8-bit signed integer
- ❑ **int**             32-bit signed integer
- ❑ **long**            64-bit signed integer
- ❑ **float**           32-bit signed floating-point value
- ❑ **double**        64-bit signed floating-point value

# More Java Data Types

## ■ Reference Types

- ❑ Refers to an object (not a primitive type)
- ❑ Can be `null` if the reference refers to nothing
- ❑ Examples: `String`, `Integer`

## ■ In Java, arrays are also reference types

```
int[] numArray;           // preferred!
```

```
int numArray[];          // also works
```

- ❑ More on arrays in a few weeks!

# Notes on Java Literals

- Boolean is simply **true** or **false**
- Integer values are straightforward
  - `int i = 17;`
- Long values use “L” suffix:
  - `long secondsInYear = 31556926L;`
  - Avoid lower-case “l” – looks like 1 in many fonts...
- Default type of a decimal value is double
  - `double pi = 3.14159265358979323;`
- Float literal uses “F” suffix:
  - `float goldenRatio = 1.618f;`
  - In this case, either “F” or “f” is fine.

# Java Character and String Literals

- Character literals can be single-quoted characters, or numbers between 0 and 65535

```
char capA = 'A';    // preferred
```

```
char capA = 65;     // harder to maintain
```

- String literals are double-quoted

```
String sandwichType = "pastrami";
```

- Special characters must be escaped:

```
String msg = "He said, \"Java is neat!\"";
```

- Most useful special characters:

`\t` = tab

`\r` = carriage return

`\n` = new line

`\\` = backslash    `\'` = single quote

`\"` = double quote



# Java Names and Naming Conventions

- Names must start with a letter, and can include only letters and digits
  - `_` and `$` are also considered “letters” in Java
  - Don’t use `$` - used by compiler for auto-generated names
- Capitalization is *very important* in Java coding style
  - Fields and methods should follow **camelCase** naming convention
  - Classes and interfaces should follow **UpperCamelCase** naming convention
  - Package names should be *all lowercase*
- Java has a number of industry-wide conventions
  - Definitely want to learn them and follow them...
  - You must follow them in CS11 Java.

# Java Variables and Initial Values

- Java variable declarations are like C/C++

```
int i;  
boolean err = false, done;  
String name = "Donnie";
```

- Local variables don't have default initial values!

```
int i;  
i = i + 1;
```

➔ Compile-time error:

**variable i might not have been initialized**

- This is an example of Java's focus on correctness
- C or C++ would compile this code without errors

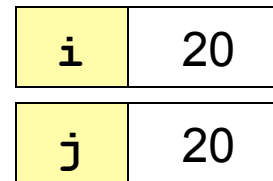
# Primitive and Reference Variables

- Difference between primitive and reference types is where the value is actually stored

- Primitive variables:

```
int i = 20;
```

```
int j = i;
```

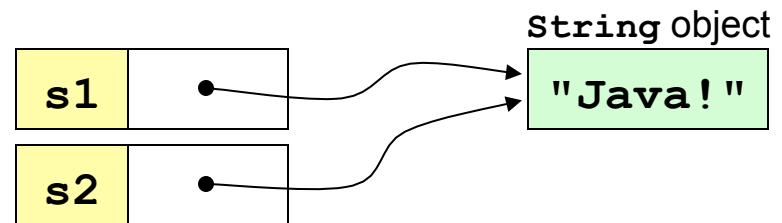


- Each variable stores its own value

- Reference variables:

```
String s1 = "Java!";
```

```
String s2 = s1;
```



- Value of reference variables is stored in main memory
- Reference variables can refer to the same object

# Java Operators

- Same set of operators as C and C++
  - Simple arithmetic: + - \* / %
  - Compound assignment: += -= \*= /= etc.
  - Increment/decrement: ++ -- (pre and post)

```
int i = 5;  
int j = ++i;           // j = 6, i = 6  
int k = i++;           // k = 6, i = 7
```
  - Comparisons: == != > >= < <=
    - Note: these operations produce **boolean** values!
    - In Java, no type can be cast to **boolean** (including **int**)
    - Also, **boolean** cannot be cast to any other type

# Logical Boolean Operators

- Again, same as C/C++: **&& || !**
  - Logical AND, logical OR, and logical NOT.
- These operators *require* **boolean** values, and *produce* **boolean** values.
- Lazy evaluation:
  - For example: `name != null && name.equals("Donnie")`
    - `name.equals(...)` only evaluated if `name != null`
  - Conversely: `name == null || !name.equals("Donnie")`
- Precedence order: **! && ||**

# String Operators

- String concatenation also uses + operator

```
public static void main(String[] args) {  
    String name = "Donnie";  
    System.out.println("Hello " + name);  
}
```

- At least one operand must be a String for + to do string-concatenation.

- + operator is evaluated left-to-right

```
int i = 5;  
int j = 4;  
System.out.println("i = " + i);    // Prints "i = 5"  
System.out.println(i + j);         // Prints "9"  
System.out.println("i + j = " + i + j); // "i + j = 54"  
System.out.println(i + j + " = i + j"); // "9 = i + j"
```

# Flow Control in Java

- Flow-control statements nearly identical to C/C++

<code>if (cond)</code>	<code>while (cond)</code>
<code>    statement;</code>	<code>    statement;</code>
<code>else if (cond)</code>	
<code>    statement;</code>	<code>do</code>
<code>else</code>	<code>    statement;</code>
<code>    statement;</code>	<code>while (cond);</code>

- Difference: *cond* must produce **boolean** value!
- Blocks of statements are enclosed with curly-braces

`{ }`, just like in C/C++

```
if (cond) {  
    statement1;  
    statement2;  
}
```

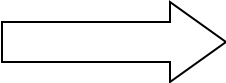
# Java For-Loops

- For loops are also very similar to C++
  - Initialize (and possibly declare) one or more looping variables
  - Test some condition before each iteration of the loop
  - Apply one or more updates to the looping variable(s)

```
for (init; condition; update) statement;  
for (init; condition; update) {  
    statement1;  
    ...  
}
```

- Equivalent to **while** loops, but more compact.

```
int i = 1;  
while (i <= 10) {  
    sum += i;  
    i++;  
}  
  
for (i = 1; i <= 10; i++)  
    sum += i;
```





# More For-Loops

- Can specify multiple initial values:

```
int i, sum;  
for (i = 1, sum = 0; i <= 10; i++)  
    sum += i;
```

- Can declare loop variables in for-loop:

```
int sum = 0;  
for (int i = 1; i <= 10; i++)  
    sum += i;
```

- In this example, `i` is only visible within the for-loop
- The *scope* of `i` is within the for-loop.

# Even More For-Loops

- Can specify multiple update operations:

```
int sum = 0;  
for (int i = 1; i <= 10; sum += i, i++) /*nothing*/;
```

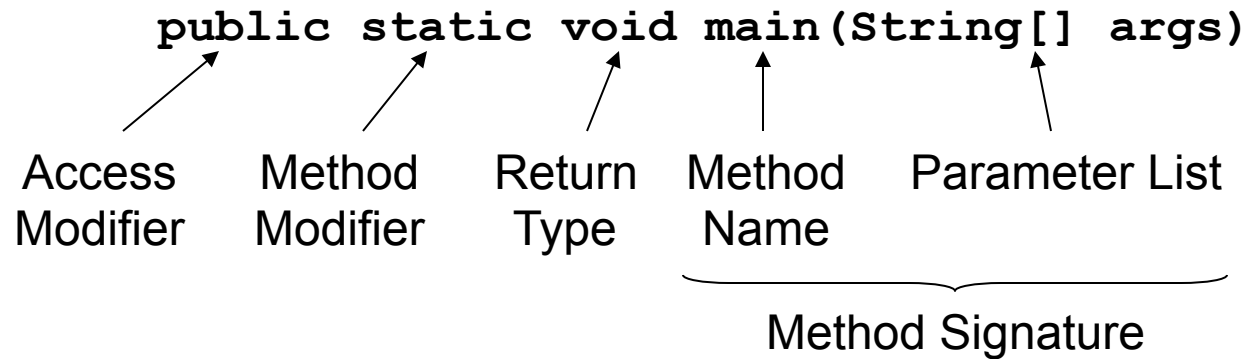
- Document that the for-loop doesn't need a body!

- Even *more* compact:

```
int sum = 0;  
for (int i = 1; i <= 10; sum += i++) /* nothing */ ;
```

- Difficult to maintain! Best to be avoided.

# Java Methods – A Brief Overview



- Methods return a value of the specified type.
- Or they return *no value*, indicated by `void` keyword.
- Methods can accept any number of arguments.
  - “No arguments” is indicated with empty parens `()`, not `void`.
- A method’s signature includes its name and its parameter-list.
- Modifiers will be covered in a bit...

# Printing in Java

- `System.out.println("Hello!");`

- Many flavors:

- `System.out.println(String x)`
  - `System.out.println(boolean x)`
  - `System.out.println(char x)`
  - `System.out.println(float x)`
  - `System.out.println(int x)`
  - `System.out.println(Object x)`
  - `System.out.println()`

- and a few more...

- These are overloaded methods.

- Same name, but different signature.

# Java Console IO

- `System.out` is the standard output stream
- `System.err` is the standard error stream
  - Use this to report errors when bad things happen.
- `System.in` is the standard input stream
  - We will use this this week.
- `System.out.println(...)` goes to next line
- Use `System.out.print(...)` to stay on same line

---

# A Note About Class Names

- Java classes can be grouped into packages
    - This is optional, but typically very helpful!
    - Packages form a hierarchy
  - **package1.package2.ClassName**
    - Package names are typically all lower-case
    - Naming rules are same as variable names.
    - Example: **java.awt.event.MouseEvent**
  - More details on this later!
-

# Terminology: Classes and Objects

- Java is *entirely* object-oriented programming (OOP) language
  - Programs are *entirely* composed of classes
- 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
- 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.

# Abstraction and Encapsulation



Key OOP  
Concepts

## ■ 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?  

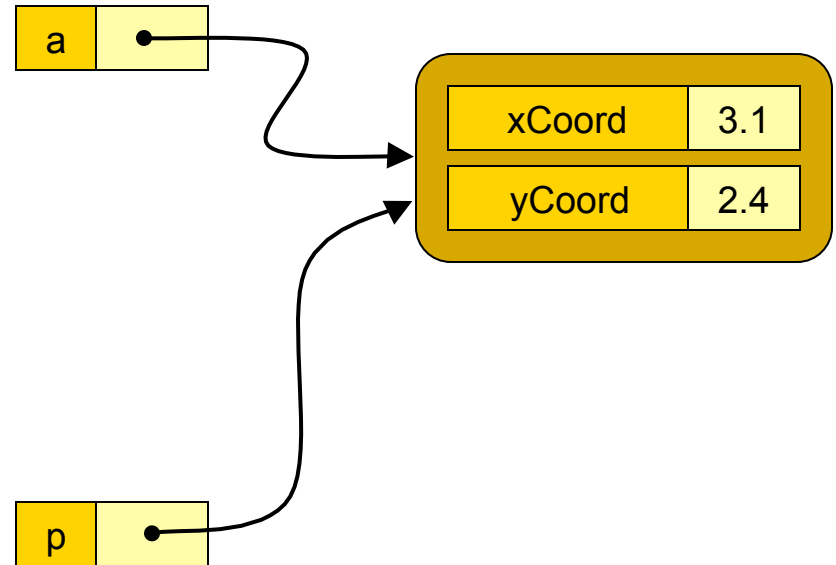
```
public void printPoint(Point2d p)
```
- Remember, **p** is a reference to the object
- Reference is copied into **p**, but the **Point2d** object that it refers to is *not*
- Side-effects and funky bugs can easily occur!



# Passing Objects in Java

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

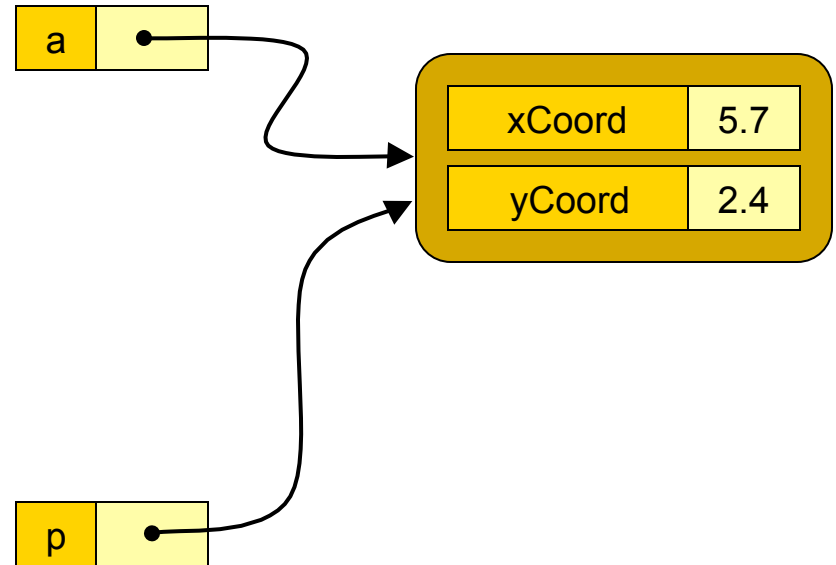
```
void printPoint(Point2d p) {  
    System.out.println(p.getX() +  
        ", " + p.getY());  
    p.setX(5.7);    // ???  
}
```



# Passing Objects in Java (2)

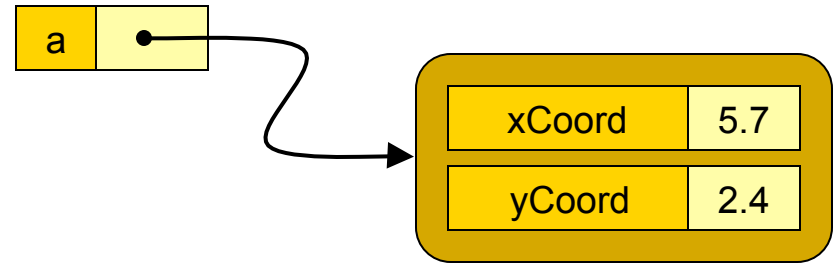
```
void main(String[] args) {  
    Point2d a =  
        new Point2d(3.1, 2.4);  
  
    printPoint(a);  
}
```

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

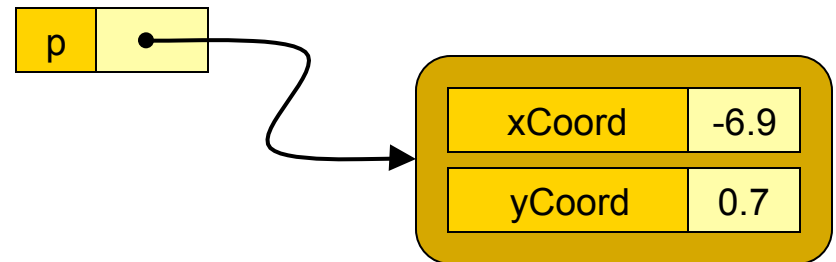


# Passing Objects in Java (3)

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

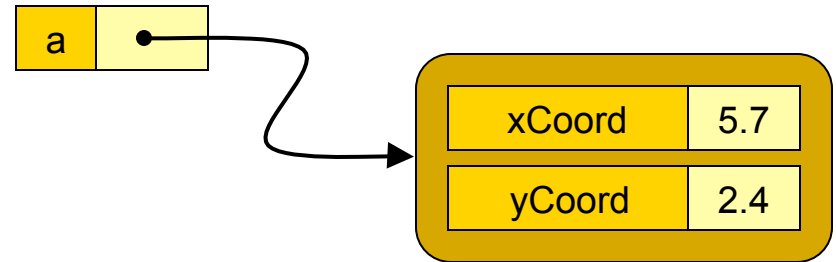


```
void printPoint(Point2d p) {  
    System.out.println(p.getX() +  
        "," + p.getY());  
    p.setX(5.7); // affects a  
    p = new Point2d(-6.9, 0.7);  
    p.setY(-2.1); // ???  
}
```

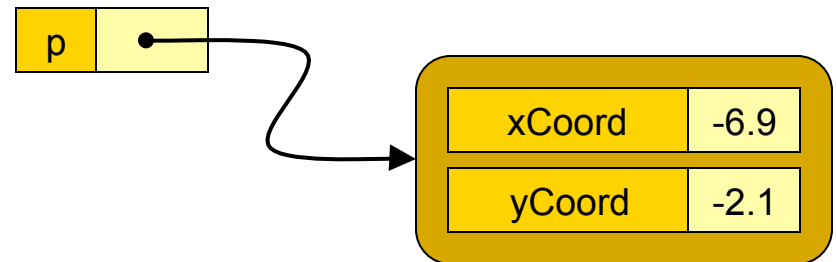


# Passing Objects in Java (4)

```
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;  
}
```

- Not an uncommon approach for mutators...
  - Argument name is same as field name
- In general, avoid unnecessary ambiguities!
  - Can lead to very subtle bugs

# 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?

**@Override**

```
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

---

# The Java API Documentation

- Complete API docs for the entire Java platform
    - Extremely useful, once you learn how to use it!
    - Auto-generated from Java library source-code
  - Lists all classes and interfaces
    - How to use them
    - What features they provide
    - Their relationships with each other
  - <http://docs.oracle.com/javase/7/docs/api/>
    - So useful, you might even want a local copy!
-

---

# Other Useful Java Documentation

- The Java Tutorial
    - Different “trails” cover different topics
    - Very helpful resource for learning new features!
  - Java Development Kit (JDK) Documentation
    - Feature-changes and new features
    - Tool documentation
  - The Java Language Specification
  - The Java VM Specification
-



---

# This Week's Homework

- Create your first Java program.
    - The CS11 object-oriented programming classic: Heron's Formula
    - Create a 3D point class, add **equals()** and **distanceTo()** methods
    - Create another class that takes 3 points as input, and computes the area of the triangle using Heron's Formula
  - Learn how to compile and run your program.
-