Today's plan

- Introduce OOP concepts from the ground up using Java
- Lots of things will be familiar from C++
- Some things will be different

```
public class Point
{
  private int x, y;
  public Point(int x, int y) {
    this.x = x; this.y = y;
  }
  public int getX() { return x; }
  public int getY() { return y; }
  public void setX(int x) { this.x = x; }
  public void setY(int y) { this.y = y; }
  public double distFromOrigin() {
    return Math.sqrt(x * x + y * y)
  }
}
```

Subclassing

- A class definition has a *superclass* (`Object` if not specified)

```
class ColorPoint extends Point { … }
```

- The superclass affects the class definition:
  - Class *inherits* all field declarations from superclass
  - Class *inherits* all private method definitions from superclass
    - Code within the subclass cannot directly access any private fields or methods.
  - But class can *override* method definitions as desired

```java
public class ColorPoint extends Point
{
  private Color color;
  public ColorPoint(int x, int y, Color c) {
    super(x, y); // call the superclass constructor
    this.color = c;
  }
  public Color getColor() { return color; }
  public void setColor(Color c) { this.color = c; }
}
```

An object has a class

```
Point p = new Point(0, 0);
ColorPoint cp = new ColorPoint(0, 0, Color.red)

/* instanceof is a keyword that returns true
   if a variable is an instance of a class. */

p instanceof Point         // true
cp instanceof ColorPoint // true
cp instanceof Point        // true
```

- Using instanceof can indicate bad OO style.
  - If you're using it to do something different for different objects types, you probably meant to write a method and have subclasses override the method.
- instanceof is an example of using reflection
  - Reflection is the ability for a computer program to be able to examine its structure and behavior at run-time.

Why subclass?

- Instead of creating **`ColorPoint`**, could add methods to **`Point`**
  - That could mess up other users and subclassers of **`Point`**

```
public class Point {
  private int x, y;
  private Color color;
  …

  public Point(x, y) {
    // what does color get set to?
  }
}
```

Why subclass?

- Instead of subclassing **`Point`**, could copy/paste the methods
  - Means the same thing *if* you don't use **`instanceof`**, but of course code reuse is nice

```
public class ColorPoint {
  private int x, y;
  private Color color;
  …
}

ColorPoint cp = new ColorPoint( whatevs )
if (cp instanceof Point) {
  // do pointy things
}
```

Why subclass?

- Instead of subclassing **`Point`**, could use a **`Point`** instance variable inside of ColorPoint.
  - Define methods to send same message to the **`Point`**
  - This is called object composition; expresses a "has a" relationship.
  - But for **`ColorPoint`**, subclassing makes sense: less work and can use a **`ColorPoint`** wherever code expects a **`Point`**

```
public class ColorPoint {
  private Point point;
  private Color color;
  public setX(int x) { point.setX(x); }
  …
}
```

Is-a vs has-a

- OO beginners tend to overuse inheritance (the is-a relationship).
- OO inheritance is notoriously tricky to get right sometimes (e.g., writing methods that test for equality)
  - boolean equals(Point a, Point b)
  - What if a & b can be Points or ColorPoints?
- Many real-world relationships can be expressed using is-a or has-a, even if the most natural way seems to be is-a.
  - ColorPoint could be written using object composition.

Circle and ellipse problem

- What should the relationship be between a Circle class and an Ellipse class?

Circle and ellipse problem

- Circles are specific types of ellipses, so a Circle is-a Ellipse.

```
public class Ellipse {
  private int radiusX, int radiusY;
  public void setRadiusX(int rx) { radiusX = rx; }
  public void setRadiusX(int rx) { radiusY = ry; }
  public int getRadiusX() { return radiusX; }
  public int getRadiusY() { return radiusY; }
}
public class Circle extends Ellipse {
  …
}
```

Circle and ellipse problem

- Circles are specific types of ellipses, so a Circle is-a Ellipse.
- But now Circle has a setRadiusX() method.
- Furthermore, what would that method's implementation look like?

Circle and ellipse problem

- Different solution: make Ellipse a subclass of Circle.
  - "An Ellipse is a Circle with an extra radius field."

```
public class Circle {
  private int radius;
  public void setRadius(int r) { radius = r; }
  public int getRadius() { return radius; }
}
public class Ellipse extends Circle {
  private int radiusY;
  // assume existing radius is for X dimension.
}
```

Circle and ellipse problem

- Different solution: make Ellipse a subclass of Circle.
  - "An Ellipse is a Circle with an extra radius field."
- Just as many problems here:

- What does it mean when an Ellipse calls Circle's setRadius or getRadius method (which radius?)

One solution: Immutability

- Let Circle inherit from Ellipse and eliminate mutator methods.

```
public class Ellipse {
  private int radiusX, int radiusY;
  public int getRadiusX() { return radiusX; }
  public int getRadiusY() { return radiusY; }
}
public class Circle extends Ellipse { … }
```

- Circle still has two radius accessor methods.
- As long as Circle's constructor forces radiusX = radiusY, there's no way to violate that constraint later.

Other solutions

- Let Circle and Ellipse inherit from some common superclass.
- Let setRadiusX() return success or failure.
- Drop inheritance entirely.
- Drop Circle; let users (manually) handle circles as instances of Ellipse.

What inheritance really is for

- Inheritance gets you into trouble when it seems like the relationship is "is-a," but it actually is "is-a-restricted-version-of."
  - Circle and Ellipse
  - Person and Prisoner
    - Certainly a Prisoner is a Person.
    - But Person can have a method walk(int distance)
    - Prisoners can't do that!
- Inheritance should be used to add extra detail to a superclass (e.g., a Monkey is an Animal), not to restrict functionality.
  - ColorPoint is (probably) fine to inherit from Point

Try this one out

- I want to declare a class ThreeDPoint.
- Should this inherit from Point?
  - What are the pros and cons?

Method overriding

- In OOP, a subclass may override a method from a superclass.
- Just re-define the method in the subclass.

In C++, what does this do?

```
class Base {
  public: int f() { return 1; } };
class Derived: public Base {
  public: int f() { return 2; } };

int main() {
  Base b;
  Derived d;
  cout << b.f() << endl;
  cout << d.f() << endl;
  b = d;
  cout << b.f() << endl;
  Base *b2 = &d;
  cout << b2->f() << endl;
}
```

```
Base *b2 = &d;
  cout << b2->f() << endl;
```

- With a pointer to an object, a call to a method of that object calls the version of the method *specified by the type of the pointer*, not the type of the object being pointed to.
- Can be changed with the C++ keyword **`virtual`**.
- With a pointer to an object, a call to a virtual method of that object calls the version of the method *specified by the type of the object being pointed to*.

In C++, what does this do?

```
class Base {
  public: virtual int f() { return 1; } };
class Derived: public Base {
  public: int f() { return 2; } };

int main() {
  Base b;
  Derived d;
  cout << b.f() << endl;
  cout << d.f() << endl;
  b = d;
  cout << b.f() << endl;
  Base *b2 = &d;
  cout << b2->f() << endl;
}
```

Java virtual methods

- In Java, all methods are virtual.
  - This behavior cannot be changed.
  - If a subclass needs to call a superclass's version of an overridden method from a subclass, there is the **`super`** keyword:

```
public class Base {
  public int f() { return 1; } }
public class Derived extends Base {
  public int f() { return 2 + super.f(); } }
```

Java virtual methods

```
public class ThreeDPoint extends Point
{
  private int z;

  // override distFromOrigin in Point
  public double distFromOrigin() {
    return Math.sqrt(
     getX()*getX() + getY()*getY() + z*z;
  }
}
```

So far…

- With examples so far, objects are not so different from closures
  - Multiple methods rather than just "call me"
  - Explicit instance variables rather than whatever is environment where function is defined
  - Inheritance avoids helper functions or code copying
  - "Simple" overriding just replaces methods

- But there is a big difference (that you learned in Java):

  Overriding can make a method define in the superclass call a method in the subclass

  - The essential difference of OOP, studied carefully next lecture

Java I/O

- Main way of outputting to the screen:
- **`System.out.println(x);`**
  - takes one argument of any type
  - if x is an object, its **`toString()`** method will be automatically called to convert it to a String.
  - also **`System.err.println(x)`**;
  - System.out is an OutputStream object (similar to **`cout`** in C++)

Java I/O

- There are about 50 bazillion ways to do input in Java.
- Easiest way:
  - **`import java.util.*;`**
  - **`Scanner scanner = new Scanner(System.in)`**
    - System.in is an InputStream object (similar to **`cin`** in C++)
  - Now call any of the following:
  - **`scanner.nextInt()`** [or nextLong(), nextFloat(), etc]
    - all of these stop at the first whitespace found
  - **`scanner.nextLine()`**
    - reads a whole line, returns a String

Try this

- Make a program that reads in integers from the keyboard until you enter -1.

Collections

- Java has many collection classes.
  - ArrayList, HashSet, HashMap most common.
  - Very few cases where you need "real" arrays; using ArrayList is much more common.

- Syntax is similar to C++ templates
  - e.g., C++'s vector, set, and map

- Gotcha: Only objects can be stored in Java's collection classes.
  - No ints, floats, booleans, doubles, etc in ArrayLists!
  - Java has "wrapper" classes Integer, Float, Boolean, Double that you use instead, and Java does the conversion for you.

ArrayList (example for ints)

- Creation
  - `ArrayList<Integer> list = new ArrayList<Integer>();`
- Put stuff in
  - `list.add(x);    // adds x to end by default`
  - `list.add(i, x); // inserts x at list[i]`
  - `list.set(i, x); // changes list[i] to x`
- Get stuff out
  - `list.get(i); // returns list[i]`
- Other stuff
  - `list.size(), list.contains(x), list.indexOf(x), list.remove(i),`

Enhanced for loop

```
for (int i = 0; i < list.size(); i++) {
  System.out.println(list.get(i));
}

for (int x : list) {
  System.out.println(x);
}
```

Try this

- Make a program that reads in integers from the keyboard until you enter -1.
- Add all the integers (as they're entered) to an ArrayList.
- Print out all the integers. Try this two ways:
  - System.out.println(list);
  - With the enhanced for loop.

Try this

- Make a program that reads in integers from the keyboard until you enter -1.
- Add a static method fib(n) that computes the n'th Fibonacci number. Write this the standard (slow, recursive) way.
- Print out the Fibonacci value of each number as they're entered.
  - What is the max Fibonacci # you can compute before you get an error?

HashMaps

- Java's has a few hashtable classes.
- Most common is HashMap.

- The Java language was constructed with hashtables in mind.
- The Object class has a hashCode() method.
  - Because all objects inherit (directly or indirectly) from Object, all classes have a hashCode() method!
- If you ever make a class that you want to use as the key of a hashtable, you should override the hashCode() and equals() methods.
  - Don't worry about this at the moment.

HashMap (example for String map to int)

- Creation
  - **`HashMap<String, Integer> map = new HashMap<String, Integer>();`**
- Put stuff in
  - **`map.put(s, i);  // associates key s with value i`**
- Get stuff out
  - **`map.get(s); // returns whatever value s is associated with`**
- Other stuff
  - **`map.size(), map.containsKey(s), map.keySet(), map.remove(s)`**

Enhanced for loop

You can use the enhanced for loop to iterate through a map:

```
for (String key : map.keySet()) {
  int value = map.get(key);
  // do something with key and/or value
}
```

Try this: memoized Fibonacci in Java

- Add a HashMap<Integer, Integer> as a static field to your class.
  - This will store the cached Fibonacci values.
- Alter your Fibonacci method so it does the following:
  - For fib(n):
  - if n = 0 or n = 1, return n
  - Check if n is a key in the hashtable.
    - If it is, get the corresponding value and return it.
    - If it's not, then
      - compute v = fib(n-1) + fib(n-2)
      - put the mapping from n to v in the hashtable
      - return v

HashSets

- A Set (ADT) is an *unordered* collection of items.
  - A List is an *ordered* collection of items.
- Java has a HashSet class that implements this ADT.
- Similar to C++'s std::set class.

HashSet (example for ints)

- Creation
  - `HashSet<Integer> set = new HashSet<Integer>();`
- Put stuff in
  - `set.add(x);      // adds x to the set`
- Test if something is in the set
  - `set.contains(x);    // returns list[i]`
- Remove something from the set
  - `set.remove(x);`
- Other stuff
  - `set.size(), set.isEmpty(), set.clear()`

And now for something completely different:

Multiple inheritance,
Java interfaces,
and abstract base classes.

More than one superclass?

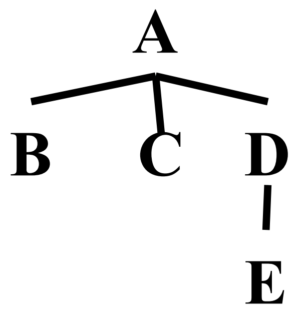
- What if we want a class that has more than one superclass?

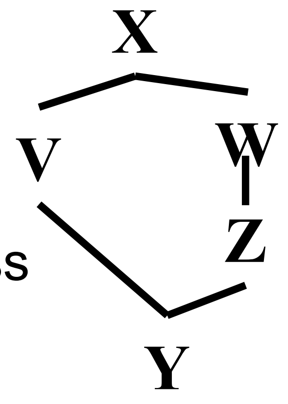
- ColorPoint3D could inherit from Point3D and ColorPoint.
- StudentAthlete inherits from Student and Athlete.

- Single inheritance can force you to use non-OOP technique to write these classes
  - (copying code or using "helper" methods)

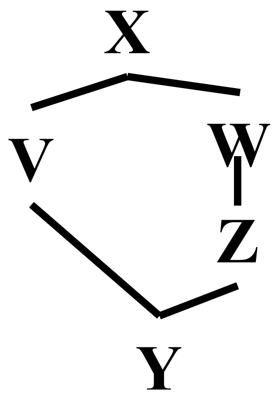
Trees, dags, and diamonds

- Note: The phrases *subclass*, *superclass* can be ambiguous
  - There are *immediate* subclasses, superclasses
  - And there are *transitive* subclasses, superclasses
- Single inheritance: the *class hierarchy* is a tree
  - Nodes are classes
  - Parent is immediate superclass
  - Any number of children allowed
- Multiple inheritance: the class hierarchy no longer a tree
  - Cycles still disallowed (a directed-acyclic graph)
  - If multiple paths show that *X* is a (transitive) superclass of *Y*, then we have *diamonds*





What could go wrong? (C++)



- If *V* and *Z* both define a method **`m`**, what does *Y* inherit? What does **`super`** mean?
  – *Directed resends* useful (e.g., **`Z.super`**)

- What if *X* defines a method **`m`** that *Z* but not *V* overrides?
  – Can handle like previous case, but sometimes undesirable (e.g., **`ColorPt3D`** wants **`Pt3D`**'s overrides to "win")

- If *X* defines fields, should *Y* have one copy of them (**`f`**) or two (**`V.f`** and **`Z.f`**)?
  – Turns out each behavior is sometimes desirable (next slides)
  – So C++ has (at least) two forms of inheritance

3DColorPoints

If Java had multiple inheritance, we would want **`ColorPt3D`** to "combine" the x and y fields into one copy of each.

```
public class Point { private int x, y; }

public class ColorPoint extends Point {
  private Color color;
}

public class Point3D extends Point {
  private int z;
}

public class ColorPoint3D extends Point, Point3D
  // not valid Java code!
```

Artistic cowboys (or cowboy-ish artists?)

This code has **`Person`** define a pocket for subclasses to use, but an **`ArtistCowboy`** wants *two* pockets, one for each **`draw`** method

```
public class Person { private Pocket pocket; }

public class Artist extends Person {
  // stores a brush in their pocket
  public void draw() { /* draw a picture */ }
}
public class Cowboy extends Person {
  // stores a gun in their pocket
  public void draw() { /* draw their gun */ }
}
public class ArtistCowboy extends Artist, Cowboy {
  // do I have one pocket, or two?
  public void draw() { /* what should I do? */ }
```

Java interfaces

- C++ has multiple inheritance (can solve the diamond problem either way you want).
- Java does not have multiple inheritance.

- Java has something similar to a classes called *interfaces*.

Java interfaces

Interfaces have no fields, only methods.
All the methods lack bodies.

```
public interface Shape {
  public double calculatePerimeter();
  public double calculateArea();
}
public class Ellipse implements Shape {
  private double radiusx, radiusy;
  public double calculatePerimeter() { … }
  public double calculateArea() { … }
}
public class Rectangle implements Shape {
  private double length, width;
  public double calculatePerimeter() { … }
  public double calculateArea() { … }
}
```

What is an interface?

```
public interface Shape {
  public double calculatePerimeter();
  public double calculateArea();
}
```

- New classes extend an existing class, but implement interfaces.
- Both classes and interfaces are types!
  - Any class that implements it is a *subtype* of it
  - So Ellipse and Rectangle are both Objects and Shapes.

```
public interface Shape {
  public double calculatePerimeter();
  public double calculateArea();
}
public class Ellipse implements Shape {
  private radiusx, radiusy;
  public double calculatePerimeter() { … }
  public double calculateArea() { … }
}
public class Rectangle implements Shape {
  private double length, width;
  public double calculatePerimeter() { … }
  public double calculateArea() { … }
}
Ellipse ell = new Ellipse();
Rectangle rect = new Rectangle();
ell instanceof Shape     // true
rect instanceof Shape    // true
ell instanceof Object    // true
rect instanceof Object   // true
```

```
public interface Shape {
  public double calculatePerimeter();
  public double calculateArea();
}
public class Ellipse implements Shape {
  private radiusx, radiusy;
  public double calculatePerimeter() { … }
  public double calculateArea() { … }
}
public class Rectangle implements Shape {
  private double length, width;
  public double calculatePerimeter() { … }
  public double calculateArea() { … }
}
Shape s1 = new Ellipse();
Shape s2 = new Rectangle();
s1 instanceof Shape     // true
s2 instanceof Shape     // true
s1 instanceof Object    // true
s2 instanceof Object    // true
```

```
Ellipse ell = new Ellipse();
Rectangle rect = new Rectangle();
Shape s1 = ell, s2 = rect;
```

/* All variables that hold objects are references (similar to pointers), so the third line above does not create new objects. */

```
double area1 = s1.calculateArea();
  // calls Ellipse's calculateArea

double area2 = s2.calculateArea();
  // calls Rectangle's calculateArea
```

/* All methods in Java are virtual, so whenever you call a method, the "correct" one is always called. */

Multiple interfaces

- Java classes can implement any number of interfaces
- Because interfaces provide no methods or fields, no questions of method/field duplication arise
  - No problem if two interfaces both require of implementers and promise to clients the same method

Summary so far

- Superclass must have fields and/or method bodies.
  - Define it as a class.
- Superclass doesn't need fields or method bodies.
  - Define it as an interface.

- What if superclass must have fields and methods,
  - but you don't know how to implement some methods in the superclass?

```
public class Shape {
  private Color color;
  public Color getColor() { return color; }
  public double calculatePerimeter() { ??? }
  public double calculateArea() { ??? }
}
public class Ellipse extends Shape {
  private double radiusx, radiusy;
  public double calculatePerimeter() { /*fine*/ }
  public double calculateArea() { /*fine*/ }
}
public class Rectangle extends Shape {
  private double length, width;
  public double calculatePerimeter() { /*fine*/ }
  public double calculateArea() { /*fine*/ }
}
```

```
public abstract class Shape {
  private Color color;
  public Color getColor() { return color; }
  public abstract double calculatePerimeter();
  public abstract double calculateArea();
}
public class Ellipse extends Shape {
  private double radiusx, radiusy;
  public double calculatePerimeter() { /*fine*/ }
  public double calculateArea() { /*fine*/ }
}
public class Rectangle extends Shape {
  private double length, width;
  public double calculatePerimeter() { /*fine*/ }
  public double calculateArea() { /*fine*/ }
}
```

Abstract classes

- Abstract classes can never be directly instantiated:

```
public abstract class X { … }
// later on
X = new X(); // nope!
```

- Can't directly instantiate interfaces either.
  - Only things that can be instantiated (new'ed) are fully-implemented classes.
- Abstract classes are a compromise between a class where all the methods are fully implemented and an interface (where none of the methods are implemented).

Examples from the Java libraries

- Comparable (and sorting)
- Number
- Collections (List, Set, Map)
- Iterable

```
for (Type i : something that implements Iterable) {
  // do stuff with i here
}
```