

Introduction to Computer Science II

CS112-2012S-23

Abstract Classes and Interfaces

David Galles

Department of Computer Science
University of San Francisco

23-0: Drawing Example

- Creating a drawing program
- Allow user to draw triangles, circles, rectangles, move them around, etc.
- Need to store a list of shapes, each of which could be a circle, rectangle, or triangle
- Shape superclass, with Triangle, Rectangle, and Circle subclasses

23-1: Shape Class

```
class Shape
{
    public void draw()
    {

    }
}
class Rectangle extends Shape
{
    public void draw()
    {
        // code to draw a Rectangle
    }
}
class Circle extends Shape
{
    public void draw()
    {
        // code to draw a Circle
    }
}
```

```
in main
-----

Shape shapes[] = new Shape[3];
shapes[0] = new Circle();
shapes[1] = new Rectangle();
shapes[2] = new Circle();

for (int i = 0; i < shapes.length; i++)
{
    shapes[i].draw();
}
```

23-2: Abstract Classes

- Abstract Class:
 - How do you draw a generic shape?
 - Drawing a generic shape doesn't make sense!
 - Does it ever make sense to instantiate a generic Shape (instead of a circle, triangle, or rectangle)?
 - No!
 - We can make the Shape class *abstract*
 - Prevents anyone from creating an instance of Shape
 - Shape *variables* are OK, as long as values are Circles, Triangles, etc

23-3: Abstract Classes

<pre>abstract class Shape { public abstract void draw(); } class Circle extends Shape { // Needs to implement draw } class Triangle extends Shape { // Needs to implement draw } class Rectangle extends Shape { // Needs to implement draw }</pre>	<pre>In main ----- Shape s1, s2, s3; // OK! s1 = new Shape(); // NOT OK! s2 = new Circle(); // OK! s3 = new Triangle(); // OK! s2.draw(); // OK! s3.draw(); // OK!</pre>
---	--

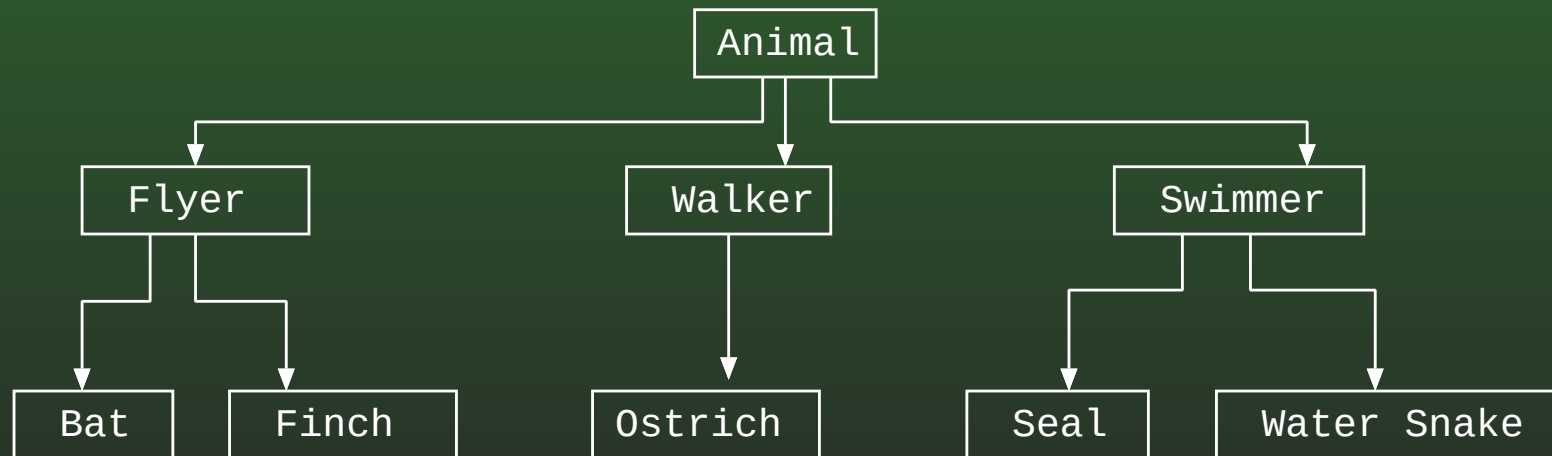
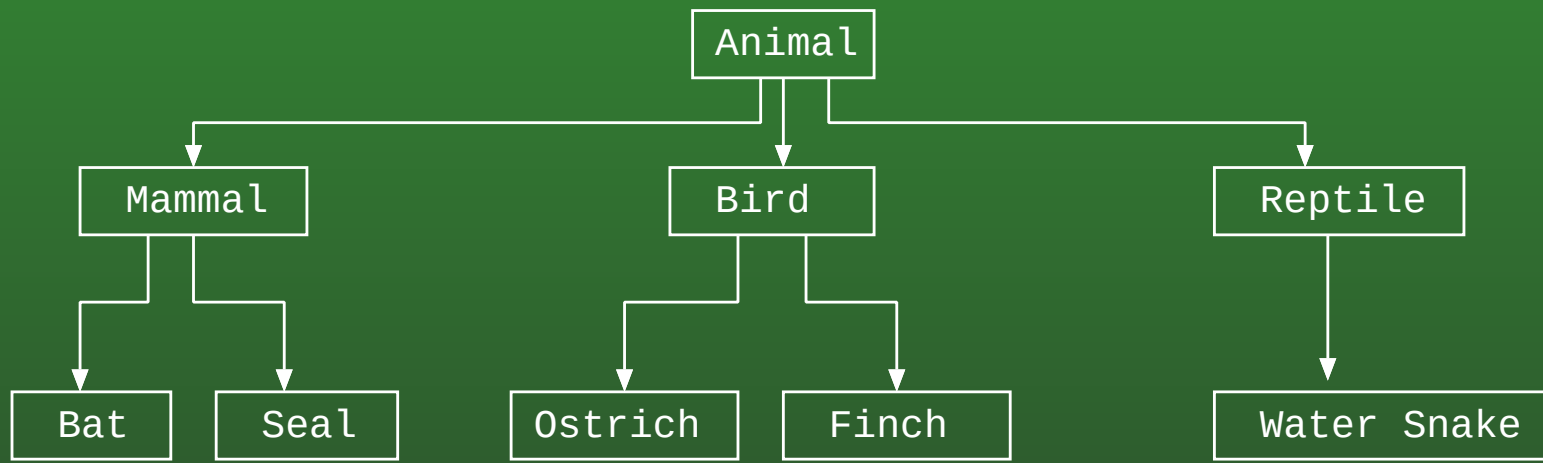
23-4: Abstract Classes

- We can make a class abstract by adding the abstract modifier to the class definition
 - Can't create instances of an abstract class
- If a class is abstract, we can define abstract methods
 - Use the abstract modifier on method definition
 - Don't give the method a body (use a ; instead of a method body)
 - Subclasses of this class will need to either implement all abstract methods, or be abstract themselves

23-5: Inheritance Heirarchies

- More than one way to skin a cat
- Classes for animals:
 - Standard classification: Mammal, bird, reptile
 - Functional Classification: Flying Animal, Swimming animal Walking Animal
- How you design your classes depends upon the problem at hand

23-6: Inheritance Heirarchies



23-7: Multiple Inheritance?

- Might be nice to inherit from more than one thing
 - Bat is a mammal *and* a flying animal
 - Could inherit both mammalian qualities, and qualities of flying animals
 - Likely want to override methods specific to bats, but would be nice to get as much “for free” as possible
- Java *does not allow* multiple inheritance
 - C++ does, however

23-8: Multiple Inheritance?

- Multiple inheritance does have problems
 - Class A defines a method foo
 - Class B also defines a method foo
 - Class C inherits from both A and B (multiple inheritance)
 - Which foo does class C use?
- Java avoids these problems by only allowing single inheritance

23-9: Interfaces

- Multiple inheritance can be useful
 - Allows more than one heirarchy structure
 - Arrange our animal classes both structurally (mammal, reptile, etc) and functionally (swims, flies, runs, etc)
- We can get some of the advantages of multiple inheritance from interfaces

23-10: Interfaces

- A java interface is essentially a promise
 - Interface defines a number of methods
 - Classes that implement the interface promise to implement all of those methods

23-11: Interfaces

```
public interface Flyer
{
    public void fly();
}
```

- Any class that implements Flyer needs to implement the fly method

23-12: Interfaces

```
public interface Flyer
{
    public void fly();
}
```

```
public class Bat extends Mammal implements Flyer
{
    public void fly()
    {
        System.out.println("I'm flying");
    }
}
```

23-13: Interfaces

```
public interface Comparable
{
    public int compareTo(Object o);
}
```

- Any class that implements Comparable needs to implement the compareTo method
- (Note – recent versions of Java use Generics in Comparable interface, but the basic idea is the same)

23-14: Interfaces

```
class Student implements Comparable
{
    public int studentID;
    public String name;

    public int compareTo(Object other)
    {
        // How can we compare an Object to a Student?
        // Really only want to compare students to other students!

        // Need a way to check if "other" is really a student
        //   If it is a student, we need to get at "student"
        //   instance variables (studentID, name)
    }
}
```


23-15: Interfaces

- instanceof:
 - (*< variable >* instanceof *<Type>*)
 - Boolean expression, returns “true” if *<variable>* is of type *<Type>*

```
Object o;  
  
// set value of o  
  
if (o instanceof String)  
{  
    // We know that o is really a String  
}  
  
if (!(o instanceof Integer))  
{  
    // We know that o is not a Integer  
}
```

23-16: Casting

- (`<Type>`) `f`
 - If `f` is not of type `<Type>`, runtime error
 - If `f` is of type `<Type>`, we can assign to a variable of type `<Type>`

```
Object o1 = new String("Hello!");  
Object o2 = new Integer(3);
```

```
String s;  
Integer i;  
i = o1;    // Not legal!   Won't even compile  
i = o2;    // Not legal!   Won't even compile  
s = o1;    // Not legal!   Won't even compile  
s = o2;    // Not legal!   Won't even compile
```

23-17: Casting

- (`<Type>`) `f`
 - If `f` is not of type `<Type>`, runtime error
 - If `f` is of type `<Type>`, we can assign to a variable of type `<Type>`

```
Object o1 = new String("Hello!");  
Object o2 = new Integer(3);
```

```
String s;  
Integer i;  
i = (Integer) o1; // Compiles, gives runtime error  
i = (Integer) o2; // Compiles & runs OK  
s = (String) o1;  // Compiles & runs OK  
s = (String) o2;  // Compiles, gives runtime error
```

23-18: Casting

- Of course, we can always assign a subclass value to a superclass variable, without casting.
 - Never gives us a runtime error
- If we assign a subclass value to a superclass variable, we can get the subclass value out of the variable by casting
 - Will give us a runtime error if the superclass variable does not hold a subclass value

23-19: Interfaces

```
class Student implements Comparable
{
    public int studentID;
    public String name;

    public int compareTo(Object other)
    {
        // Following will cause runtime exception if we try to
        //   compare a Student with a non-student.
        int otherID = ((Student) other).studentID;
        if (studentID < otherID)
            return -1;
        else if (studentID > otherID)
            return 1;
        else
            return 0;
    }
}
```

23-20: Using Interfaces

- We can declare a variable of type “Comparable”
- Can assign any comparable value to type Comparable

```
Comparable c1, c2, c3;  
c1 = new Student();  
c2 = Integer(4);    // Integer class implements Comparable  
c3 = "Hello";       // String class does, too
```

23-21: Using Interfaces

- Creating a comparable variable seems a little silly
- However, write a function that takes a Comparable variable as a parameter makes perfect sense
- Even better, a function that takes an array of Comparable objects as an input parameter

23-22: Using Interfaces

- Write a method that takes as input an array of “Comparable”
 - That is, we can pass in an array of anything, as long as elements of that array implement Comparable
- Returns the smallest element in the array

23-23: Using Interfaces

```
Comparable minValue(Comparable array[])  
{  
}
```

- Return the smallest element in the array
- If the array is empty, return null

23-24: Using Interfaces

```
Comparable minValue(Comparable array[])
{
    if (array.length == 0)
        return null;
    Comparable smallest = array[0];
    for (int i = 1; i < array.length; i++)
    {
        if (array[i].compareTo(smallest) < 0)
            smallest = array[i];
    }
    return smallest;
}
```

23-25: Using Interfaces

```
Integer intArray[] = new Integer[10];  
// fill up intArray with Integers
```

```
Student studentArray[] = new Student[20];  
// fill up student array with Students
```

```
Integer smallestInteger = smallest(intArray); // BAD!! Why?  
Student smallestStudent = smallest(studentArray); // BAD!! Why?
```

23-26: Using Interfaces

```
Integer intArray = new Integer[10];  
// fill up intArray with Integers
```

```
Student studentArray = new Student[20];  
// fill up student array with Students
```

```
Integer smallestInteger = (Integer) smallest(intArray);  
Student smallestStudent = (Student) smallest(studentArray);
```

23-27: Sorting

- Want to sort an array of integers
 - Break the list into a sorted portion and an unsorted portion
 - Repeatedly insert the next element in the unsorted portion of the list into the sorted portion of the list (examples on board)

23-28: Sorting

```
public static void sort(Comparable data[])
{
    for (int i = 1; i < data.length; i++)
    {
        Comparable nextElem = data[i];
        int j;
        for (j=i-1; j >= 0 && data[j].compareTo(nextElem) > 0; j--)
        {
            data[j+1] = data[j];
        }
        data[j+1] = nextElem;
    }
}
```

23-29: Sorting

- This sorting method can sort any array of comparables
 - Integers
 - Strings
 - Students
 - ... anything that implements the comparable interface