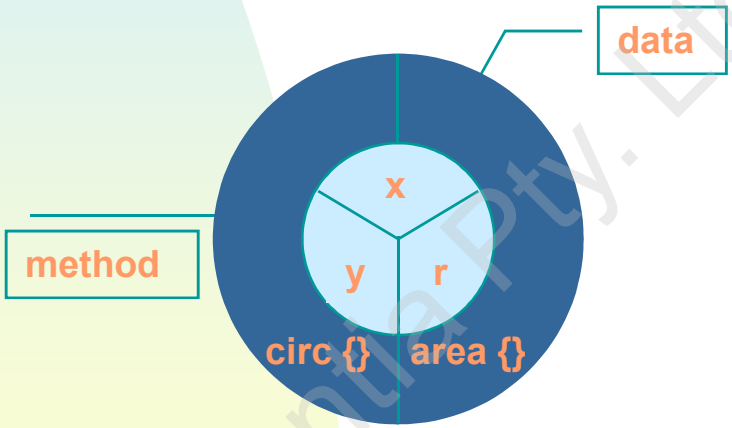
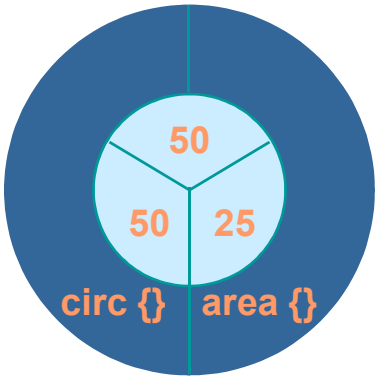
OOP With Java

- Class == abstraction
  - concept, real-world ‘thing’, etc.
  - also a packaging mechanism
- Object == concrete instance of a class



Circle class == abstract

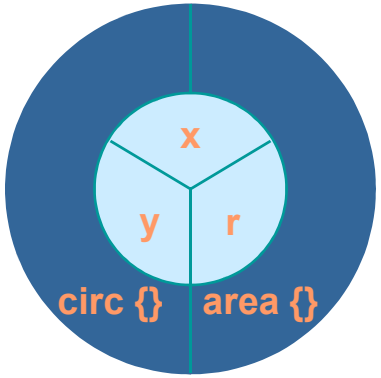


Circle object == concrete

OOP With Java

- Class Circle
  - a class is an encapsulation of data & methods

```
public class Circle extends Shape
 {
 public double x,
               y,
               r;

 public double circ ()
   {
   return (2 * (Math.PI * r));
   }

 public double area ()
   {
   return (Math.PI * (r * r));
   }
 }
```



  - An object is an instance of a class
    - created by applying the new operator to the class' constructor

```
{
Circle c = new Circle ();
c.x = c.y = 50.0; c.r = 2.4;
double theArea = c.area ();
```

constructor

method call

OOP With Java

- Constructors
  - have the same name as the class
  - no return type specified; type implied
  - can be > 1 for different situations:

```
public class Circle extends Shape
  {
  …
  public Circle ()
    {  // create a default circle
    this (0.0, 0.0, 1.0);
    }

  public Circle (double x, double y, double r)
    {…}; // create with given properties
  }
```

calls this constructor

'this' is a reference to the enclosing instance

OOP With Java

- statement that invokes ‘this’ must be the first in a constructor
- objects will get a default constructor
  - if not explicitly provided
  - numeric fields -> 0; references -> null

- No destructors
  - Object finalization dealt with differently
    - less satisfactorily?

OOP With Java

- Object destruction
  - no destructors as in C++
  - Java uses *garbage collection*
    - objects are reference counted
    - automatic reclamation
    - mark large objects (e.g. big arrays no longer needed) for early reclamation by setting reference to null

```
public static void main (String [] args)
  {
  double [] array = new double [100000];
  doSomethingWith (array);
  array = null;
  …
```

    - can invoke System.gc () directly

OOP With Java

- GC calls a finalizer

```
protected void finalize () throws IOException
  {
  if (filedescr != null)
    close ();

  super.finalize ();
  }
```

  - invoked just before space is taken back
  - no guarantees
    - GC may never be invoked; no specified order, etc.
  - finalizer may ‘resurrect’ an object
    - by saving the this reference somewhere ‘safe’
    - BAD style!
  - exceptions are ignored by runtime

OOP With Java

- Method *overloading*
  - methods with same name
    - int f (int);
    - double f (double)

    } overloaded
  - based on a method's signature
    - [name, parameters (#, type)]

    note: return type not considered
  - saw this before: constructors

```
public void print (int I);
public void print (long l);
public void print (char [] s);
public void print (String s);
public void print (boolean b);
…
```

OOP With Java

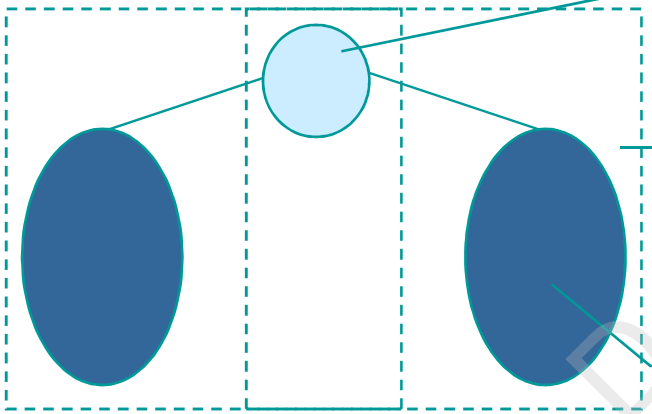
- Primitive types passed by value *only* (ie Copies)
  - references are primitive types!
  - swap (*X*, *X*) *not possible*
    - solution: pass elements via reference to array

```
public class CanSwap
  {
  private static final int APOS = 0, BPOS = 1;

  private static void swap (final int [] a)
    { int t = a [APOS]; a [APOS] = a [BPOS]; a [BPOS] = t; }

  public static void main (final String [] args)
    {
    int theInt0 = 0, theInt1 = 1;
    // flashback to BASIC!!!
    int [] intArray = {theInt0, theInt1};
    swap (intArray);
    theInt0 = intArray [APOS]; theInt1 = intArray [BPOS];
    }
  }
```

OOP With Java

- Class/instance distinctions
  - instance methods/data
    - 1 copy / object
  - class methods/data
    - introduced by ‘static’ keyword
    - 1 copy shared between all instances
    - methods have no ‘this’ reference
      - can only act on parameters and class data
    - closest Java gets to globals
      - but no possibility of name ‘clashes’

OOP With Java



class cannot be extended

```
public final class MyMath
   {
   public static final int PI = 3;

   public static double square (final double n)
      {
      return (n * n);
      }
   …
   }

   {
   double d = MyMath.square (MyMath.PI);
```

allocated *once* only

class name, not instance name

OOP With Java

- Static initialisers
  - [multiple] unnamed arbitrary code blocks
    - multiple instances amalgamated according to source-code order
  - run once when class is *first* loaded

```
public final class MyMath
  {
  static private double [] sines = new double [1000];

  // set up sine lookup table for speed
  static
    {
    double x = 0.0,
           dx = (Math.PI / 2) / (1000 - 1);

    for (int i = 0; i < 1000; i ++)
      { sines [i] = Math.sin (x); x += dx; }
    }
  }
```

OOP With Java

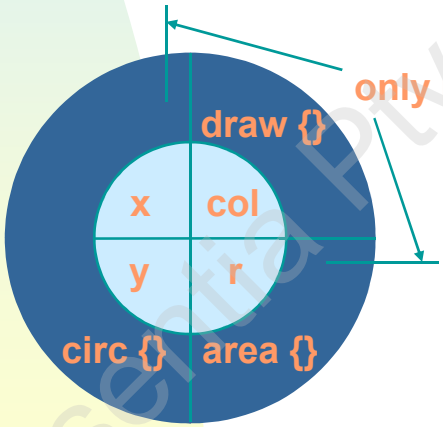
- Inheritance
  - main impetus: enhance code reuse and quality

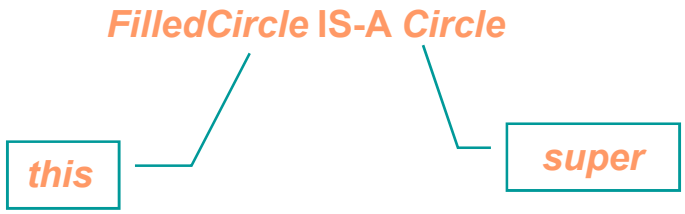
```
public class FilledCircle extends Circle
  {
  ...
  }
```

  - everything extends Object
    - even if nothing explicitly stated
    - default methods
      - clone
      - equals
      - toString
      - getClass
      - etc.

OOP With Java

- models an IS-A relationship
  - a circle IS-A shape, which IS-A object
    - a circle HAS-A x coordinate (which IS-A float)
- allows complex structures to be built up through 'diffs'



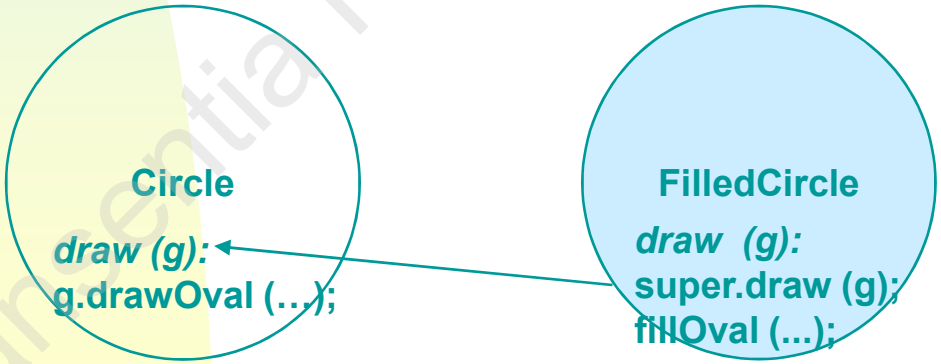


OOP With Java

```
public class FilledCircle extends Circle
  {
  private Color col;

  public FilledCircle (float x, float y, float r, Color col)
    {
    super (x, y, r);
    this.col = col;
    }

  public void draw (Graphics g)
    {
    super.draw (g);
    …;   // do the special FilledCircle 'magic'
    }
  }
```

OOP With Java

- The ‘super’ keyword
  - refers to immediate ‘ancestor’: “super.name *is treated exactly as if it had been the expression* ((S)this).name*; thus, it refers to the field named* name *of the current object, but with the current object viewed as an instance of the superclass.”*
  - super.super.x illegal
    - use ((*SomeSuperClass*) this).x explicitly



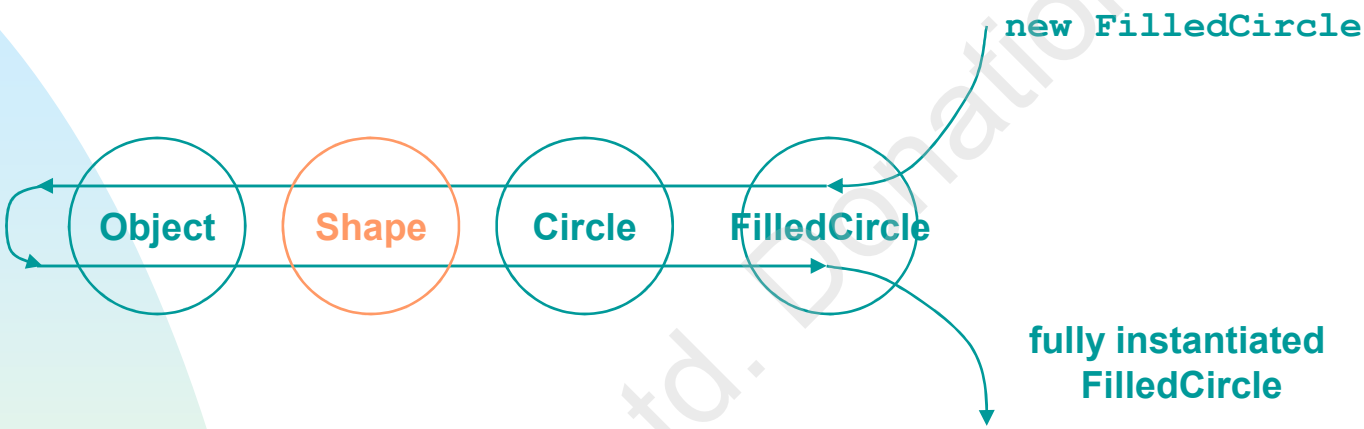
OOP With Java

```
class T0
  {
  public int f () { return (0); }
  }
class T1 extends T0
  {
  public int f ()
    {
    System.out.println ("\t" + super.f ());
    return (1);
    }
  }
class T2 extends T1
  {
  public int f ()
    {
    System.out.println ("\t\t" + super.f ());
    return (2);
    }
  }
public class Super
  {
  public static void main (String [ ] args)
    {
    System.out.println (new T2 ().f ());
    }
  }
```

```
>java Super
        0
                1
2
```

OOP With Java

- Constructor chaining



  - Java doesn't do automatic finalize chaining
    - should call super.finalize () explicitly at end
- Final classes
  - may not be extended
    - (also final methods)

OOP With Java

```
public class Chain extends Super
  {
  public Chain ()
    {
    System.out.println ("Making a Chain...");
    }

  public static void main (String [] args)
    {
    new Chain ();
    }
  }

class Super extends SuperSuper
  {
  public Super ()
    {
    System.out.println ("Making a Super...");
    }
  }

class SuperSuper
  {
  public SuperSuper ()
    {
    System.out.println ("Making a SuperSuper...");
    }
  }
```

```
C:\Chain>java Chain
Making a SuperSuper...
Making a Super...
Making a Chain...

C:\Chain>
```

OOP With Java

- Overriding
  - method with same signature as in superclass
  - overriding != overloading
  - allows “Dynamic Dispatching” / “Late Binding”
    - Square, Circle extends Shape
    - call to myShapeVariable.area ()
      - whether circle or square’s area called depends on *dynamic* type referenced by myShapeVariable
        - all Java methods ≈ C++’s virtual by default
        - (possibly) except if final
    - can’t cast behaviour away
      - don’t want programmer to be able to force call to square’s area instead of circle’s

OOP With Java

- Polymorphism/upcasting
  - taking an object reference and treating it as a reference to a less specialised type in the inheritance tree
  - useful for “generic” structures

```
import java.util.*;
public class Poly
  {
  public static void main (String [] args)
    {
    Vector v = new Vector ();
    v.addElement (new Circle (2));
    v.addElement (new Square (2));
    Enumeration e = v.elements ();
    while (e.hasMoreElements ())
      {
      Shape s = (Shape) e.nextElement ();
      if (s.area () > 2)
        s.draw ();
      }
    }
  }
```

OOP With Java

```
interface Drawable { public void draw (); }

abstract class Shape implements Drawable
  {
  public abstract double area (),
                         circ ();
  public abstract void draw ();
  }

class Circle extends Shape
  {
  int radius;
  public Circle (int radius) { this.radius = radius; }
  public double circ () { return (2 * Math.PI * radius); }
  public double area () { return (Math.PI * Math.pow (radius, 2)); }
  public void draw () { … }
  }

class Square extends Shape
  {
  int lengthOfSide;
  public Square (int lengthOfSide) { this.lengthOfSide = lengthOfSide; }
  public double area () { return (lengthOfSide * lengthOfSide); }
  public double circ () { … }
  public void draw () { … }
  }
```

OOP With Java

- Run-time type info.
  - the class java.lang.Class
    - one instance for each loaded class
      - created by runtime system
    - describes a class
      - forName (String) returns info for a name
      - isInterface
      - getName
      - getSuperclass
      - newInstance
      - etc.

OOP With Java

- reflecting upon ‘this’ allows an object to find out it’s name...

```
import java.io.*;

public class SimpleReflectionTest
  {
  public SimpleReflectionTest()
    {
    System.out.println (this.getClass ().getName ());
    }

  // recall that main is a member of the class,
  // not an instance, so make an instance of
  // SimpleReflectionTest to reflect upon
  public static void main (String [] args)
    {
    new SimpleReflectionTest();
    }
  }
```

OOP With Java

- Encapsulation
  - controlling visibility
    - packages
    - visibility modifiers
    - technique: getters & setters
  - key: *don't give away info. unless you absolutely have to*

OOP With Java

- Packages
  - namespace of related (cooperating?) classes
    - java.awt.Frame; java.awt.event.ActionEvent
    - *not* hierarchical, contrary to appearance
    - au.edu.dstc.bobs.stuff.MyClass.doSomething ()

  - package au.edu.dstc.bobs.stuff;
    - *first* thing in file
  - proposed naming convention:
    - bob@mach.net.dom -> dom.net.package.class….
    - should get unique package names this way

OOP With Java

- ∃ a default, unnamed package
  - classes belong to this unless otherwise positioned
- in JDK, name defines filesystem directory structure
  - bobs.stuff.MyClass ➔ bobs/stuff/MyClass.class
  - not an absolute requirement
    - e.g. VisualAge for Java has a repository instead
- CLASSPATH environment variable
  - location of user-defined components
  - location of system components appended automagically
  - not made available to appl{ications, ets}

OOP With Java

- import
  - 2 forms
    - import package.*class*;
    - import package.*;
  - import provided as a convenience only
    - could use fully-qualified names everywhere
  - possible ambiguities resolved by using fully-qualified names
- java.lang.* always implicitly imported

OOP With Java

- Visibility modifiers
  - accounts for packages as well as classes
  - unspecified (“friendly”)
    - visible throughout its enclosing package
  - public
    - (class/interface) visible anywhere the containing package is; (method) visible anywhere it’s class is
  - private
    - only visible within it’s own class
  - protected
    - visible throughout package and any extensions to the containing class

OOP With Java

- must only be 1 public class per file
  - rest are friendly throughout the package
  - act in a “supporting role”
  - so: need to consider class, file and package boundaries
- Visibility summary

	Member Visibility			
Accessible to:	**Public**	**Protected**	**Package**	**Private**
Same class	🗐	🗐	🗐	🗐
Class in same package	🗐	🗐	🗐	⧖
Subclass in different package	🗐	🗐	⧖	⧖
Non-subclass, different package	🗐	⧖	⧖	⧖

OOP With Java

- Techniques
  - getters/setters
    - secure but slightly cumbersome

```
public class Weekday
  {
  private int weekday;        // restricted to MONDAY .. SUNDAY
  public static final int MONDAY = 0,
                          TUESDAY = MONDAY + 1,
                          …
                          SUNDAY = SATURDAY + 1;
  public void setWeekday (int weekday) throws Exception
    {
    if ((weekday < MONDAY) || (weekday > SUNDAY))
      throw new Exception ("Day of week out of range.");
    this.weekday = weekday;
    }
  public int getWeekday ()
    {
    return (weekday);
    }
  }
```

    - required technique for JavaBeans

OOP With Java

- Abstract classes
  - a description of what is needed
    - provides a ‘shape’ to guide construction *within the inheritance hierarchy*
  - “fill in the blanks”
    - any class with an abstract method must be declared as such
    - cannot be instantiated
    - a subclass becomes concrete only if it overrides *all* abstract methods
    - if a subclass leaves abstract methods, it is itself abstract

OOP With Java

```
abstract class Shape implements Drawable
  {
  public abstract double area (),
                         circ ();
  public abstract void draw ();
  }

class Circle extends Shape
  {
  int radius;
  public Circle (int radius) { this.radius = radius; }
  public double area () { return 2 * Math.PI * radius; }
  public double circ () { … }
  public void draw () { … }
  }
```

OOP With Java

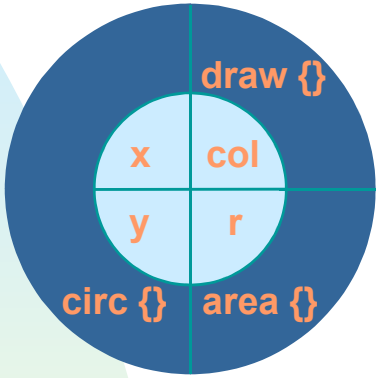
- Interfaces
  - collection of method *definitions* and constants
    - every field in an interface is implicitly public, static and final
    - every method is implicitly public
  - useful for capturing similarities *without an inheritance relationship*:
    - DrawableCircle = Circle + [Drawable I/F]
    - DrawablePerson = Person + [Drawable I/F]
    - obviously, Circle & Person are not related via inheritance

OOP With Java

```
// Drawable.java
public interface Drawable
  {
  double DEFAULT_X = 0.0,
         DEFAULT_Y = DEFAULT_X;
  void setColor (Color c);
  void setPosition (double x, double y);
  void draw (DrawWindow dw);
  }


// DrawableCircle.java
public class DrawableCircle extends Circle implements Drawable
  {
  Color theColor;
  …
  // this method (partially) fulfils the Drawable 'promise'
  public void setColor (Color c)
    {
    theColor = c;
    }

// DrawablePerson.java
public class DrawablePerson extends Person implements Drawable
  {
  Color theColor;
  …
  // this method (partially) fulfils the Drawable 'promise'
  public void setColor (Color c)
    {
    theColor = c;
    }
```

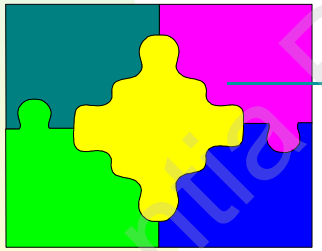
OOP With Java

- consider a FilledCircleButton
  - can't inherit from both FilledCircle *and* Button…



+ active: responds to user interaction

interface: a promise that the class will implement certain methods



interfaces allow an object to fit in where required

  - so: also useful for enhancing single inheritance chain

OOP With Java

- interface defines a new type: use accordingly

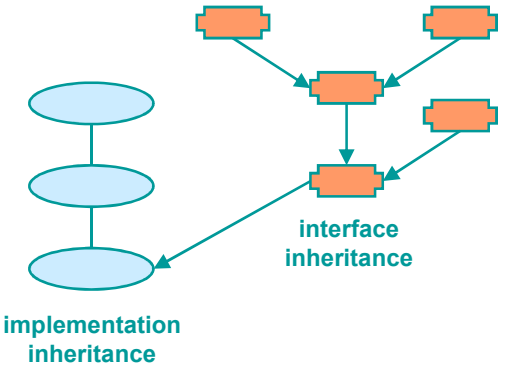
  "A variable whose declared type is an interface type may have as its value a reference to any object that is an instance of a class declared to implement the specified interface. It is not sufficient that the class happen to implement all the abstract methods of the interface; the class or one of its superclasses must actually be declared to implement the interface…"

OOP With Java

- a class can implement > 1 interface
- an interface can extend 1+ interfaces

```
public interface Transformable extends Scalable, Rotatable, Reflectable { }
public interface DrawingObject extends Drawable, Transformable { }

public class Shape implements DrawingObject { }
Or...
public class Shape implements Drawable, Transformable { }
```

- interface defines a 'protocol' for using the implementation
  - e.g. callbacks



OOP With Java

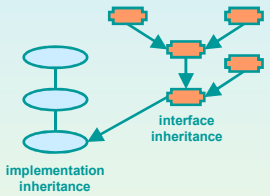
```
// Timer.java
class Timer implements Runnable
  {
  private int interval = 0;
  privaate Timed target;

  public Timer (Timed target, int interval)
    {
    this.target = target;
    this.interval = interval;
    Thread.currentThread ().setDaemon (true);
    }

  public void run ()
    {
    for ( ; ; )
      {
      try
        {
        Thread.sleep (interval);
        }
      catch (InterruptedException ie)
        {
        ie.printStackTrace (System.err);
        }
      target.tick ();
      }
    }
  }
```

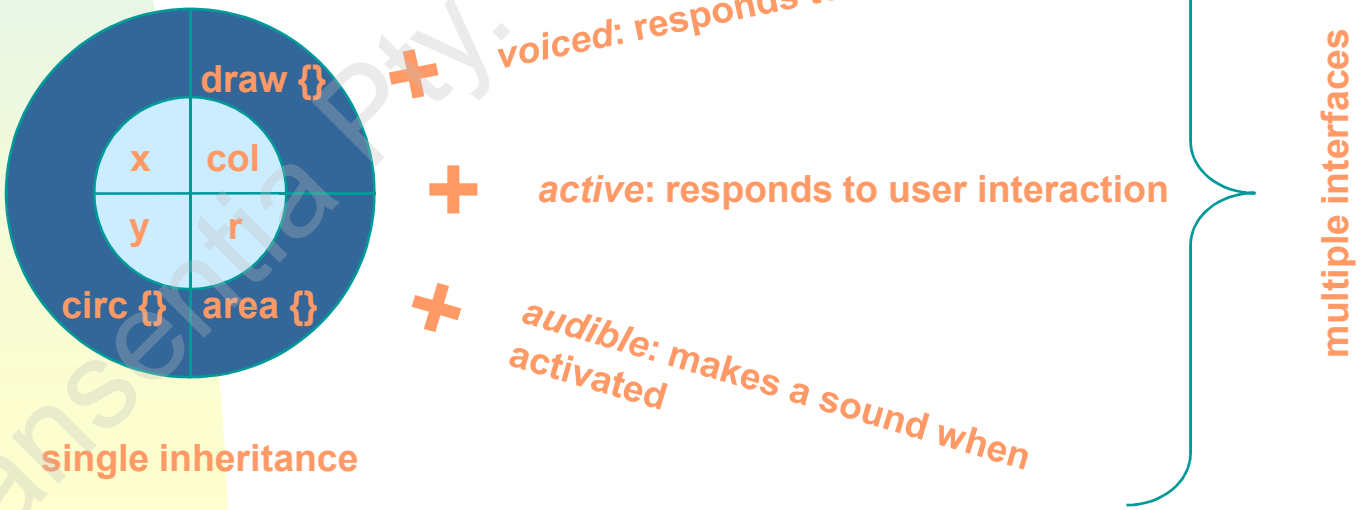
```
// ClockCanvas.java
class ClockCanvas extends Canvas
  implements Timed
  {
  public ClockCanvas (int interval)
    {
    new Thread (new Timer (this, interval)).start ();
    }

  public void tick ()
    { /* do something */ }
  }
```

```
// Timed.java
interface Timed
  {
  public void tick ();
  }
```

OOP With Java

- Interfaces facility ≈ multiple inheritance?
  - M.I. a Bad Thing
    - complex compiler & ghastly disambiguating rules
  - interfaces???
    - simpler: no code inherited, just definitions
    - more cumbersome?





OOP With Java

- lesson from “swing” classes:
  - abstract classes allow for changes in a class definition
    - use if a system is still felt to be immature and still subject to change
    - changes in abstract class permeates all subclasses without effort
  - interface fixes changes for all time
    - appropriate to ‘fix’ a mature class hierarchy
    - would be troublesome to re-define all classes affected by change in interface definition