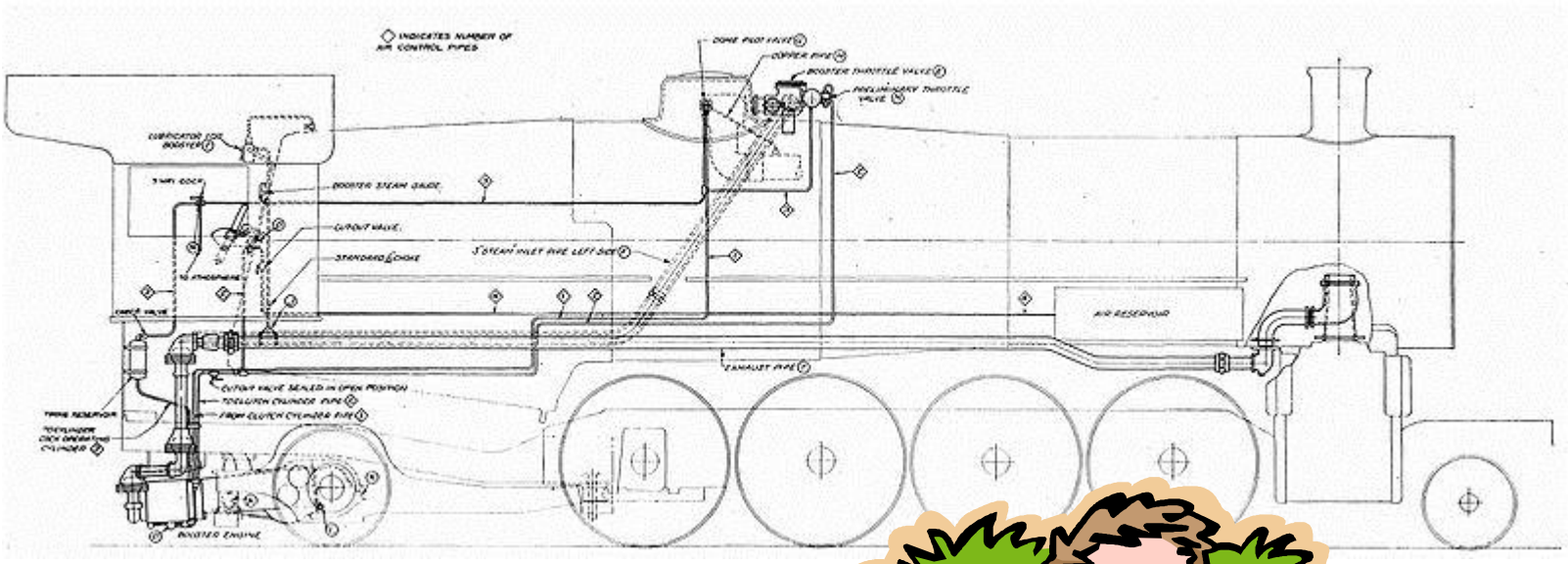
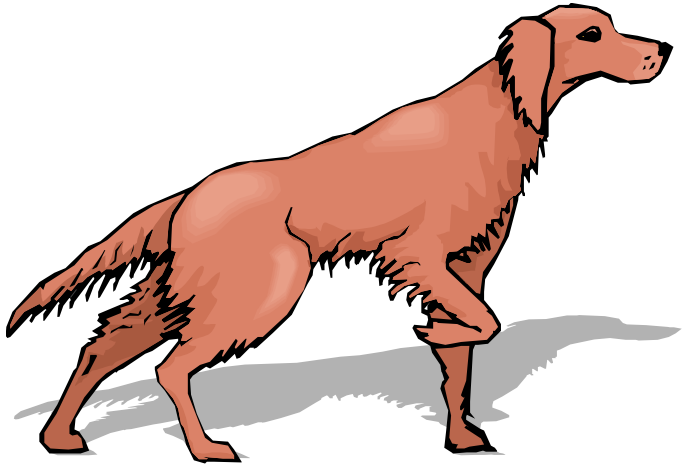
OOP in Java





Overview

- Object Oriented Programming (OOP) in Java
  - Review of core constructs
- Namespaces in Java
  - Package and import statement
- Data encapsulation
  - Access modifiers
- Inheritance
  - Polymorphism
  - Abstract base classes vs. concrete classes
  - Interfaces

Namespaces

- Complex software:
  - Often uses many small classes
  - Problem: multiple classes with same name
    - e.g. `List` is in `java.awt` and `java.util`
- Namespace
  - Container for a set of identifiers (names)
    - Programmer uses prefixes to select specific container
  - Declare package name at top of each class
    - `package com.keithv;`
    - Source lives in com/keithv subdirectory
  - Others can import one or all of package's classes
    - `import com.keithv.*;`

Data encapsulation

- Data encapsulation
  - Hides implementation details of an object
  - Clients don't have to care about details
  - Allows class designer to change implementation
    - Won't break previously developed clients
  - Provides convenient location to add debug code
  - Don't expose implementation details
    - Use `private` access modifier

Access modifiers

- Access modifier
  - All instance variables and methods have one
    - `public` - everybody can see/use
    - `private` - only class can see/use
    - `protected` - class, subclasses outside package, everybody else in package (!)
    - `default` - everybody in package, what you get if you don't specify a access modifier

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

  - Normally:
    - Instance variables: `private`
    - Methods world needs: `public`
    - Helper methods used only inside the class: `private`

Inheritance

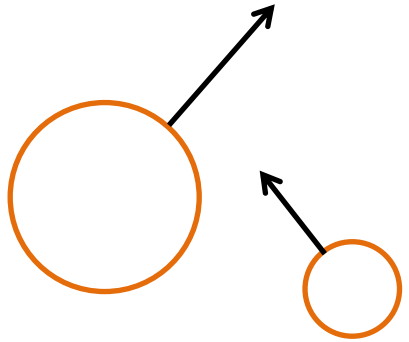
- One class can "extend" another
  - Parent class: shared vars/methods
  - Child class: more specific vars/methods
    - Children extend their parent

- Lets you share code
  - Repeated code is evil

- Store similar objects in same bucket
  - Can lead to simpler implementations

Inheritance example

- Goal: Animate circles that bounce off the walls
  - What does an object know?
    - x-position, y-position
    - x-velocity, y-velocity
    - radius
  - What can an object do?
    - Draw itself
    - Update its position, check for bouncing off walls

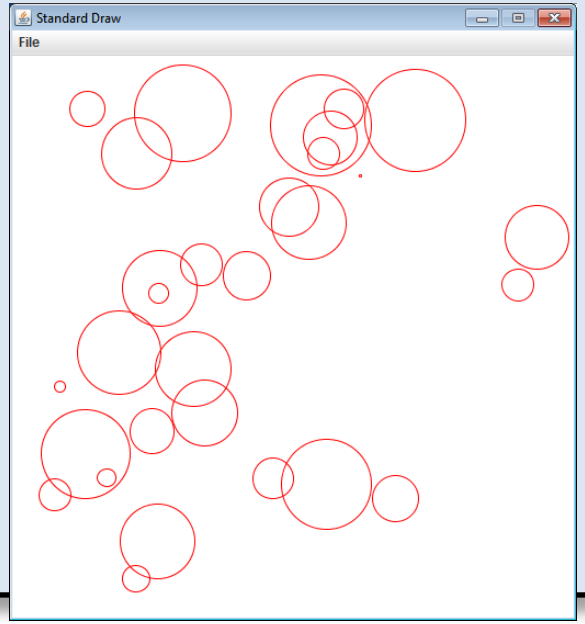


Bouncing circle class

```
public class Circle
{
   private double x, y, vx, vy, r;

   public Circle(double x, double y, double vx, double vy, double r)
   {
      this.x  = x;
      this.y  = y;
      this.vx = vx;
      this.vy = vy;
      this.r  = r;
   }
   public void draw()
   {
      StdDraw.setPenColor(StdDraw.RED);
      StdDraw.circle(x, y, r);
   }
   public void updatePos()
   {
      x += vx;
      y += vy;
      if ((x < 0.0) || (x > 1.0))
         vx *= -1;
      if ((y < 0.0) || (y > 1.0))
         vy *= -1;
   }
}
```

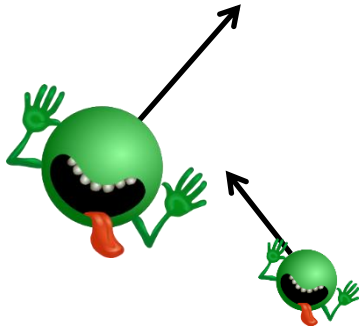
Bouncing circle client

```
public class CircleClient
{
   public static void main(String[] args)
   {
      Circle [] circles = new Circle[30];
      for (int i = 0; i < circles.length; i++)
         circles[i] = new Circle(Math.random(),
                                 Math.random(),
                                 0.002 - Math.random() * 0.004,
                                 0.002 - Math.random() * 0.004,
                                 Math.random() * 0.1);
      while (true)
      {
         StdDraw.clear();
         for (int i = 0; i < circles.length; i++)
         {
            circles[i].updatePos();
            circles[i].draw();
         }
         StdDraw.show(10);
      }
   }
}
```



Inheritance example

- Goal: Add images that bounce around
  - What does an object know?
    - x-position, y-position
    - x-velocity, y-velocity
    - radius
    - image filename
  - What can an object do?
    - Draw itself
    - Update its position, check for bouncing off walls



```
public class CircleImage
{
   private double x, y, vx, vy, r;
   private String image;

   public CircleImage(double x, double y, double vx, double vy, double r,
                      String image)
   {
      this.x     = x;
      this.y     = y;
      this.vx    = vx;
      this.vy    = vy;
      this.r     = r;
      this.image = image;
   }
   public void draw()
   {
      StdDraw.picture(x, y, image, r * 2, r * 2);
   }
   public void updatePos()
   {
      x += vx;
      y += vy;
      if ((x < 0.0) || (x > 1.0))
         vx *= -1;
      if ((y < 0.0) || (y > 1.0))
         vy *= -1;
    }
}
```

All this code appeared in the Circle class!

Inheritance: bouncing circular images!

This class is a child of the `Circle` class

```
public class CircleImage extends Circle
{
   private String image;   // image representing this object

   public CircleImage(double x, double y, double vx, double vy, double r,
                      String image)
   {
      super(x, y, vx, vy, r);
      this.image = image;
   }

   public void draw()
   {
      StdDraw.picture(getX(), getY(), image, getRadius() * 2, getRadius() * 2);
   }
}
```

Calls the `Circle` constructor which sets all the other instance variables.

Overridden version of `draw()` method, this one draws a picture scaled according to the radius.

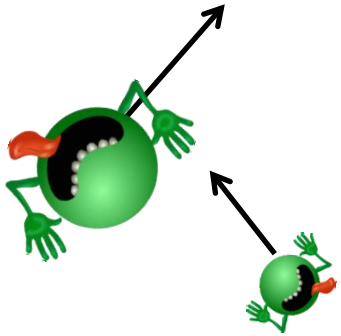
NOTE: Need getter methods to get at instance variables declared in parent.

Override = method with same method signature as parent's method

Overload = multiple methods in same class with different signatures

Inheritance example

- Goal: Add images that bounce and rotate
  - What does an object know?
    - x-position, y-position
    - x-velocity, y-velocity
    - radius
    - image filename
    - rotation angle
  - What can an object do?
    - Draw itself
    - Update its position, check for bouncing off walls, rotate image by one degree



Rotating bouncing circular image class

```
public class CircleImageRotate extends CircleImage
{
   private int angle;   // current rotation angle of image

   public CircleImageRotate(double x, double y, double vx, double vy, double r,
                            String image)
   {
      super(x, y, vx, vy, r, image);
   }

   public void draw()
   {
      StdDraw.picture(getX(), getY(), getImage(),
                      getRadius() * 2, getRadius() * 2, angle);
   }

   public void updatePos()
   {
      angle = (angle + 1) % 360;
      super.updatePos();
   }
}
```

Calls the constructor of our parent class `CircleImage`.

Calls the `updatePos()` in our parent's parent class `Circle`.

Client with three object types

- Goal: Random collection of bouncing circles, images and rotating images
- Without inheritance:
  - Create three different arrays (tedious!)

```
Circle            [] circles1 = new Circle[10];
CircleImage       [] circles2 = new CircleImage[10];
CircleImageRotate [] circles3 = new CircleImageRotate[10];
```

  - Fill in all three arrays (tedious)
  - Loop through them separately (tedious!)

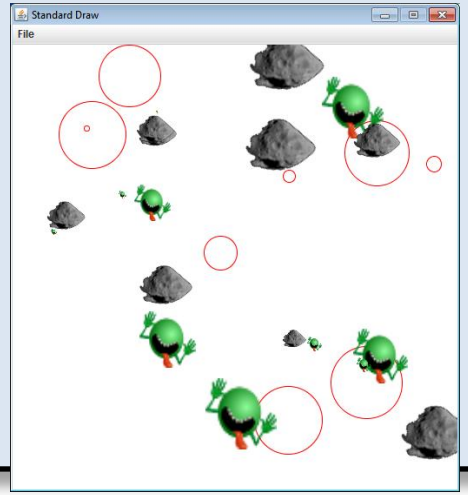
```
for (int i = 0; i < circles1.length; i++)
   circles1[i].updatePos();

for (int i = 0; i < circles2.length; i++)
   circles2[i].updatePos();

for (int i = 0; i < circles3.length; i++)
   circles3[i].updatePos();
```

Client with three object types

```
Circle [] circles = new Circle[30];
for (int i = 0; i < circles.length; i++)
{
   int rand  = (int) (Math.random() * 3.0);
   double x  = Math.random();
   double y  = Math.random();
   double vx = 0.002 - Math.random() * 0.004;
   double vy = 0.002 - Math.random() * 0.004;
   double r  = Math.random() * 0.1;

   if (rand == 0)
      circles[i] = new Circle(x, y, vx, vy, r);
   else if (rand == 1)
      circles[i] = new CircleImage(x, y, vx, vy, r, "dont_panic_40.png");
   else
      circles[i] = new CircleImageRotate(x, y, vx, vy, r, "asteroid_big.png");
}

while (true)
{
   StdDraw.clear();
   for (int i = 0; i < circles.length; i++)
   {
      circles[i].updatePos();
      circles[i].draw();
   }
   StdDraw.show(10);
}
```

With inheritance:

Put them all together in one array!



What method gets run?

```
while (true)
{
    StdDraw.clear();
    for (int i = 0; i < circles.length; i++)
    {
      circles[i].updatePos();
      circles[i].draw();
    }
    StdDraw.show(10);
}
```

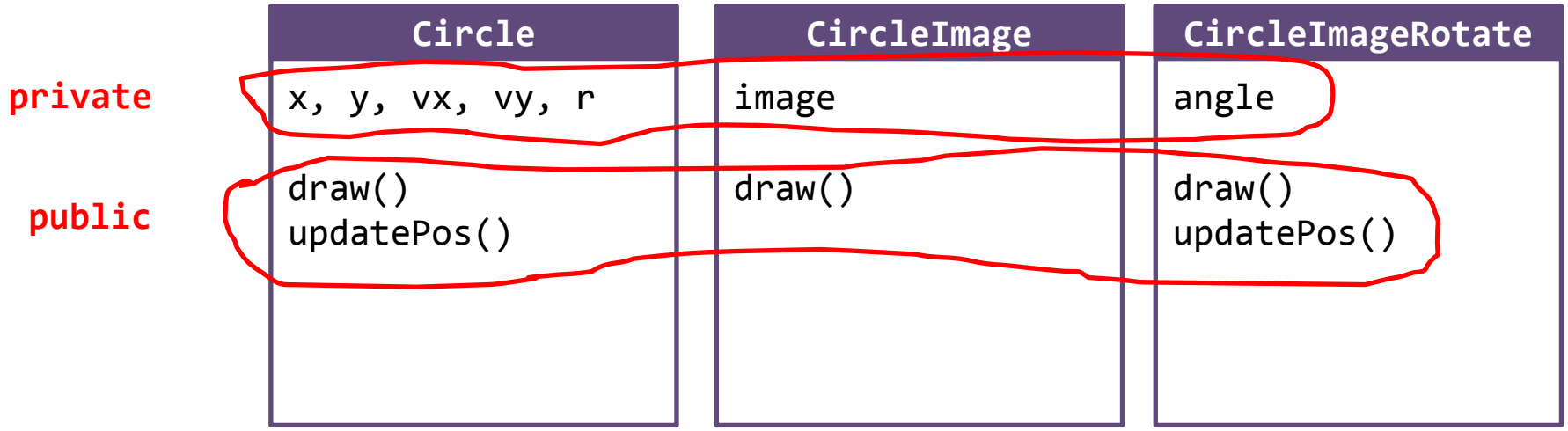
circles[i] could be:
Circle, CircleImage or
CircleImageRotate object

Circle	CircleImage	CircleImageRotate
x, y, vx, vy, r draw() updatePos()	image draw()	angle draw() updatePos()

Most specific method runs. If the subclass has the desired method, use that. Otherwise try your parent. If not, then your parent's parent, etc.

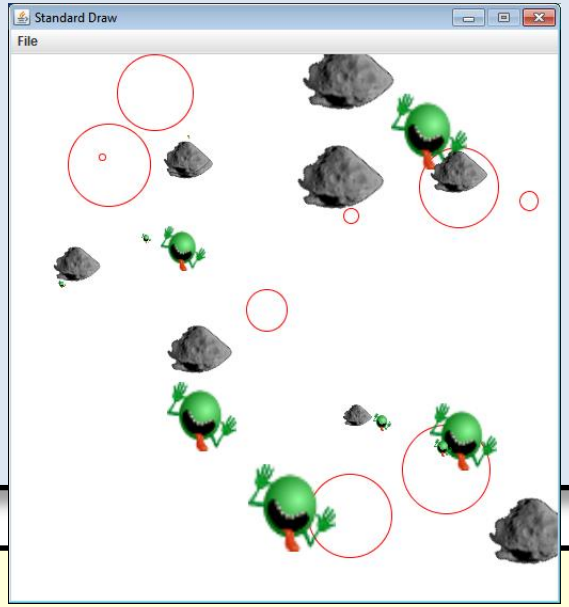
Access modifiers

- Access modifiers
  - Controls if subclasses see instance vars/methods
    - `private` = only the class itself
    - `public` = everybody can see
    - no modifier (default) = everybody in package
    - `protected` = everybody in package, any class that extends it (even if outside package)



Simplified main program

```
Bouncers bouncers = new Bouncers();

for (int i = 0; i < 30; i++)
   bouncers.add();

while (true)
{
   StdDraw.clear();
   bouncers.updateAll();
   bouncers.drawAll();
   StdDraw.show(10);
}
```




```
public class Bouncers
-----------------------------------------------------------------------
void add()        // add a random type of bouncing object with a
                  // random location, velocity, and radius
void updateAll()  // update the position of all bouncing objects
void drawAll()    // draw all the objects to the screen
```

Application Programming Interface (API) for the Bouncers class.

Bouncer implementation, 1/2

```
public class Bouncers
{
   private ArrayList<Circle> objs = new ArrayList<Circle>();

   public void add()
   {
      int rand  = (int) (Math.random() * 3.0);

      double x  = Math.random();
      double y  = Math.random();
      double vx = 0.002 - Math.random() * 0.004;
      double vy = 0.002 - Math.random() * 0.004;
      double r  = Math.random() * 0.1;

      if (rand == 0)
         objs.add(new Circle(x, y, vx, vy, r));
      else if (rand == 1)
         objs.add(new CircleImage(x, y, vx, vy, r, "dont_panic_40.png"));
      else
         objs.add(new CircleImageRotate(x, y, vx, vy, r, "asteroid_big.png"));
   }
...
```

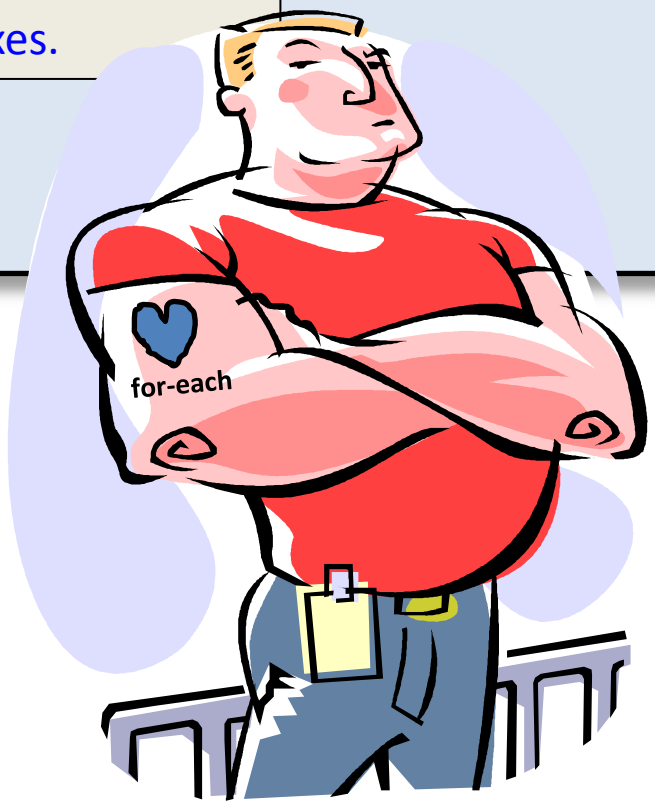
I decided to use an `ArrayList` as my underlying data structure, but clients of `Bouncers` don't know and don't have to care.

Bouncer implementation, 2/2

```
    public void updateAll()
    {
        for (Circle obj : objs)
            obj.updatePos();
    }

    public void drawAll()
    {
        for (Circle obj : objs)
            obj.draw();
    }
}
```

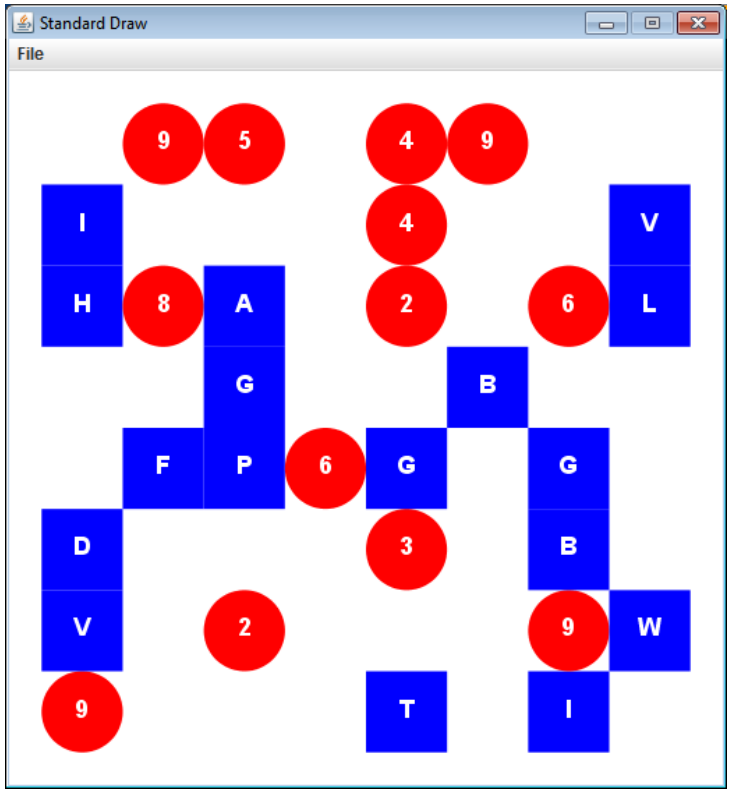
Perfect time to bust out the enhanced for loop.

Much more succinct than looping over all the integer indexes.



A tile game

- Goal: Design classes for use in a tile game
  - Played on a square N x N grid
  - Each grid location can have one thing:
    - Square letter tile
    - Circular number tile
  - Some other rules: TBD

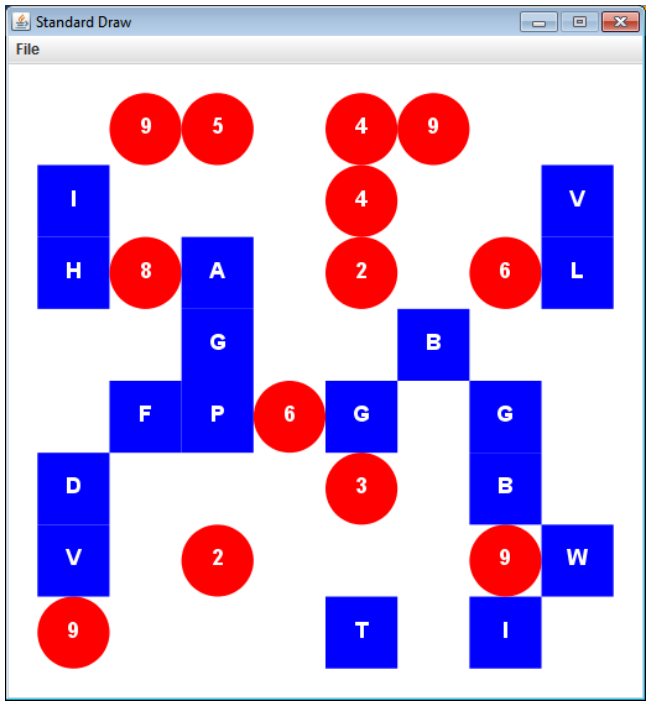


Designing the Tile game

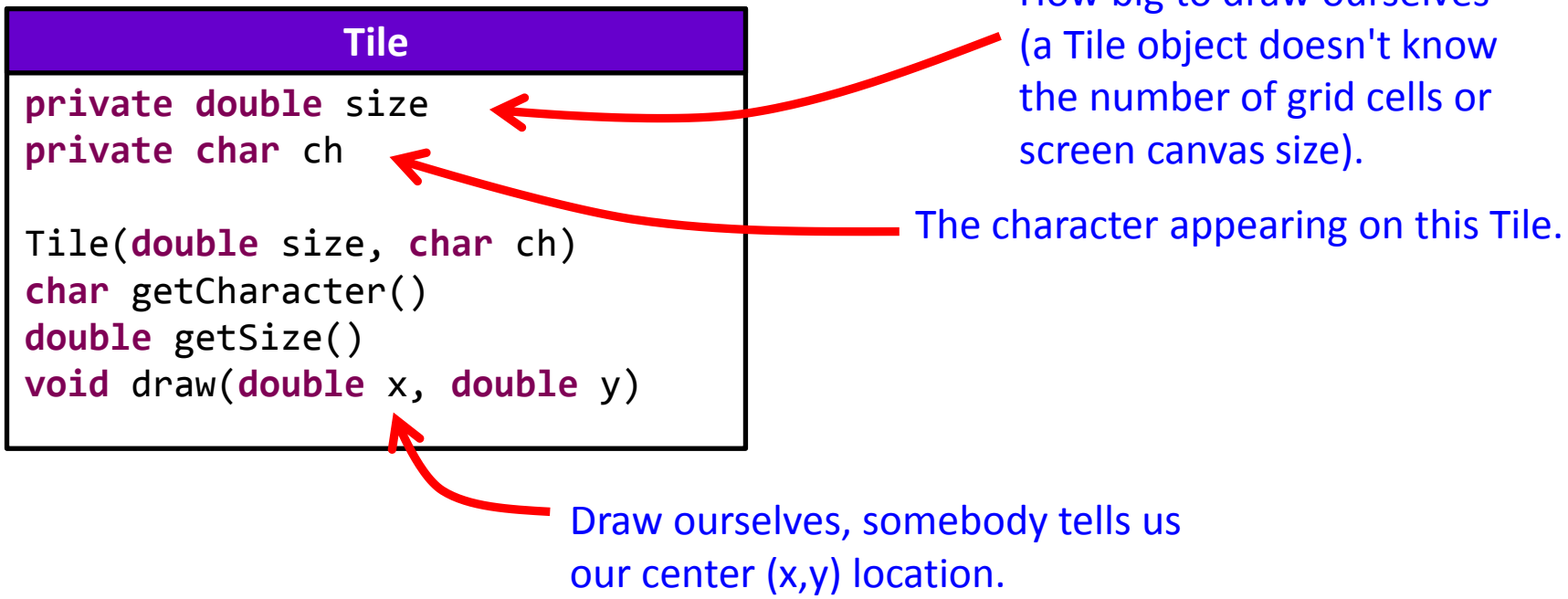
- Use a 2D array for the N x N grid
  - Array element `null` if no tile there
  - Otherwise reference to letter/number tile object
  - Start by randomly placing non-overlapping tiles

```
final int GRID  = 8;
Tile [][] tiles = new Tile[GRID][GRID];
```

null	null	null	null	null	null	null	null
null	null	null	null	null	null	null	null
null	null	null	null	null	null	null	null
null	null	null	null	null	null	null	null
null	null	null	null	null	null	null	null
null	null	null	null	null	null	null	null
null	null	null	null	null	null	null	null
null	null	null	null	null	null	null	null



Single class design?

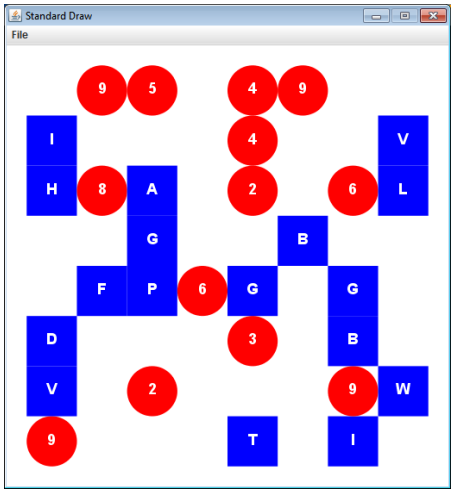


- Problem: `draw()` has to check character to know how to draw itself
  - Any method whose behavior depends on tile type needs similar conditional logic

Tile class hierarchy

Tile

```
private double size
private char ch

Tile(double size, char ch)
char getCharacter()
double getSize()
void draw(double x, double y)
```



LetterTile extends Tile

```
LetterTile(double size)
void draw(double x, double y)
```

Draw a square tile

Construct using a random letter A-Z

NumberTile extends Tile

```
NumberTile(double size)
void draw(double x, double y)
```

Draw a circle tile

Construct using a random number 0-9

Terminology: Concrete vs. Abstract class

- Concrete class
  - Some classes make sense to create
  - e.g. `LetterTile`, `NumberTile`
- Abstract class
  - Some classes don't make sense to create
  - Exist so children can inherit things
  - Exist so related objects can live in same array
  - e.g. `Tile`

Abstract `Tile` class

Prevents anyone from creating a Tile object

Child classes will get these methods for free

```
public abstract class Tile
{
   private double size = 0.0;
   private char   ch   = '\0';

   public Tile(double size, char ch)
   {
      this.size = size;
      this.ch = ch;
   }

   public char getCharacter() { return ch;  }
   public double getSize()    { return size; }

   public abstract void draw(double x, double y);

}
```

All child classes must implement a method called draw with exactly this signature.

This is what makes the polymorphism work (i.e. we can put any child of `Tile` into the same array and call `draw()` on any element).

Concrete LetterTile class

```
public class LetterTile extends Tile
{
   public LetterTile(double size)
   {
      super(size, (char) StdRandom.uniform((int) 'A', (int) 'Z' + 1));
   }

   public void draw(double x, double y)
   {
      StdDraw.setPenColor(StdDraw.BLUE);
      StdDraw.filledRectangle(x, y, getSize() / 2.0, getSize() / 2.0);

      StdDraw.setPenColor(StdDraw.WHITE);
      StdDraw.text(x, y, "" + getCharacter());
   }
}
```

Randomly assigns a letter between A and Z

Since we extend Tile, we must implement all abstract methods declared in Tile

Concrete NumberTile class

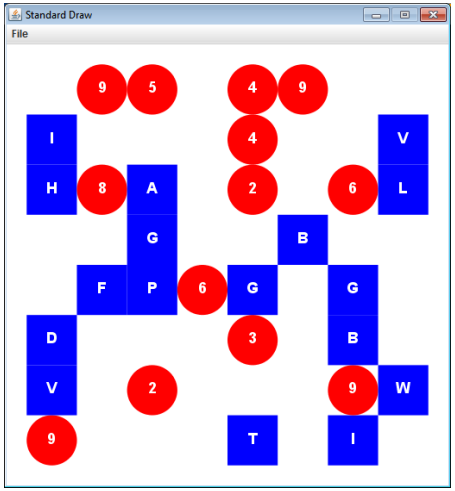
```
public class NumberTile extends Tile
{
   public NumberTile(double size)
   {
      super(size, (char) StdRandom.uniform((int) '0', (int) '9' + 1));
   }

   public void draw(double x, double y)
   {
      StdDraw.setPenColor(StdDraw.RED);
      StdDraw.filledCircle(x, y, getSize() / 2.0);

      StdDraw.setPenColor(StdDraw.WHITE);
      StdDraw.text(x, y, "" + getCharacter());
   }
}
```

Randomly assign a letter between 0 and 9

Since we extend `Tile`, we must implement all abstract methods declared in `Tile`

TileBoard class

- Manage the N x N grid inside another class
  - Create for a given number of tiles, grid size, and canvas size
  - Draw itself



TileBoard

```
private Tile [][] tiles // 2D array storing LetterTile and NumberTile objs
private double size     // Size of tiles in StdDraw coordinates

TileBoard(int numTiles, int gridSize, double canvasSize)
void draw()
```

TileBoard constructor

```
public TileBoard(int numTiles, int gridSize, double canvasSize)
{
   tiles = new Tile[gridSize][gridSize];
   size = canvasSize / gridSize;

   int added = 0;

   // Keep adding tiles until we reach the target number or board limit
   while ((added < numTiles) && (added < gridSize * gridSize))
   {
      // Choose a random (x, y) grid location for the next tile
      int x = (int) (Math.random() * gridSize);
      int y = (int) (Math.random() * gridSize);
      if (tiles[x][y] == null)
      {
         // Randomly choose a letter or number tile
         if (Math.random() < 0.5)
            tiles[x][y] = new LetterTile(size);
         else
            tiles[x][y] = new NumberTile(size);
         added++;
      }
   }
}
```

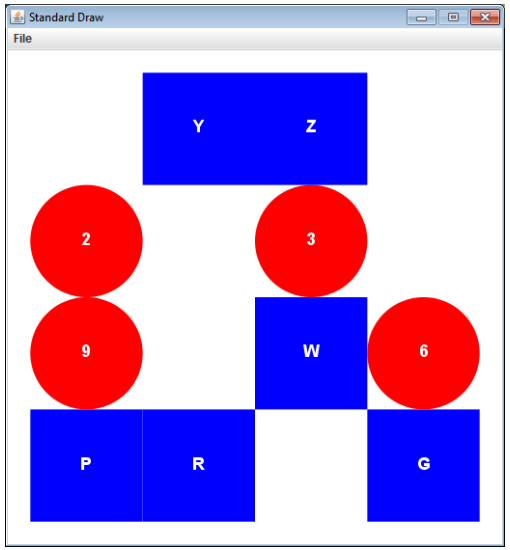
TileBoard drawing

```
public void draw()
{
   StdDraw.setFont(new Font("SansSerif", Font.BOLD, 18));

   // Loop over all the x-locations in the grid
   for (int x = 0; x < tiles.length; x++)
   {
      // Loop over all the y-locations in this x row
      for (int y = 0; y < tiles[x].length; y++)
      {
         // Only draw if the grid location contains a tile
         if (tiles[x][y] != null)
             tiles[x][y].draw(size * (x + 0.5), size * (y + 0.5));
      }
   }
}
```

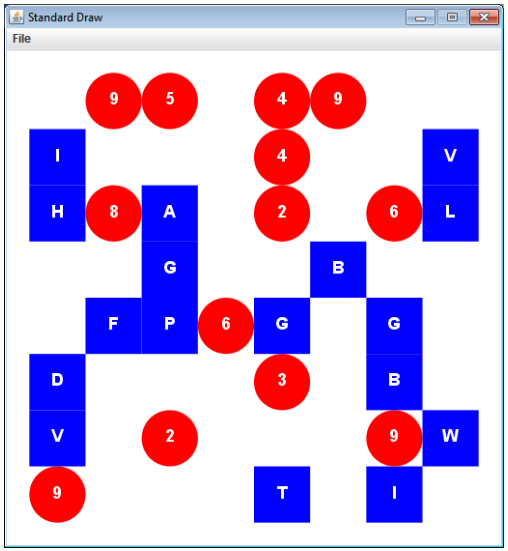
TileGame main program

```
public class TileGame
{
   public static void main(String [] args)
   {
      TileBoard board = new TileBoard(Integer.parseInt(args[0]),
                                      Integer.parseInt(args[1]),
                                      1.0);

      board.draw();
   }
}
```

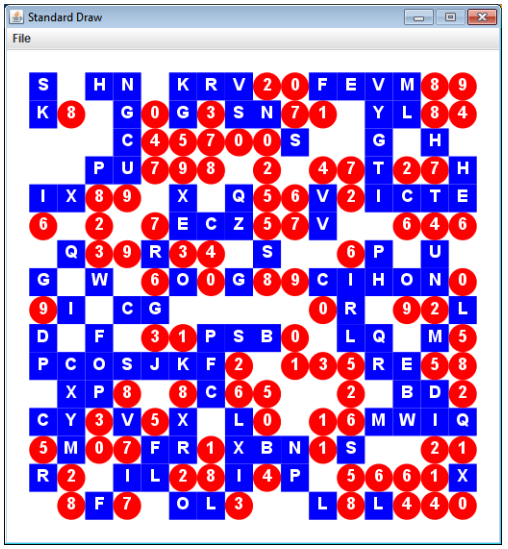
```
% java TileGame 10 4
```



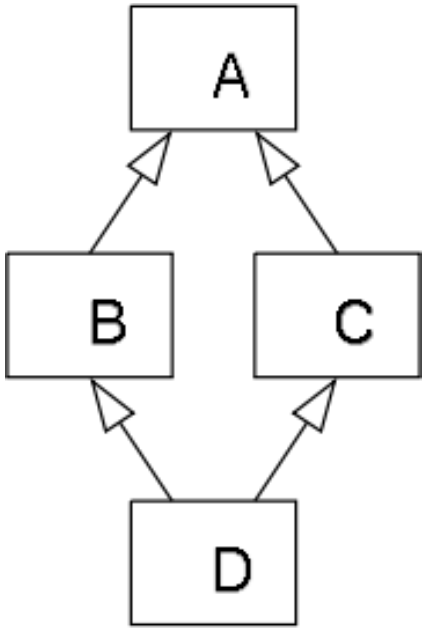

```
% java TileGame 30 8
```




```
% java TileGame 200 16
```



Interfaces

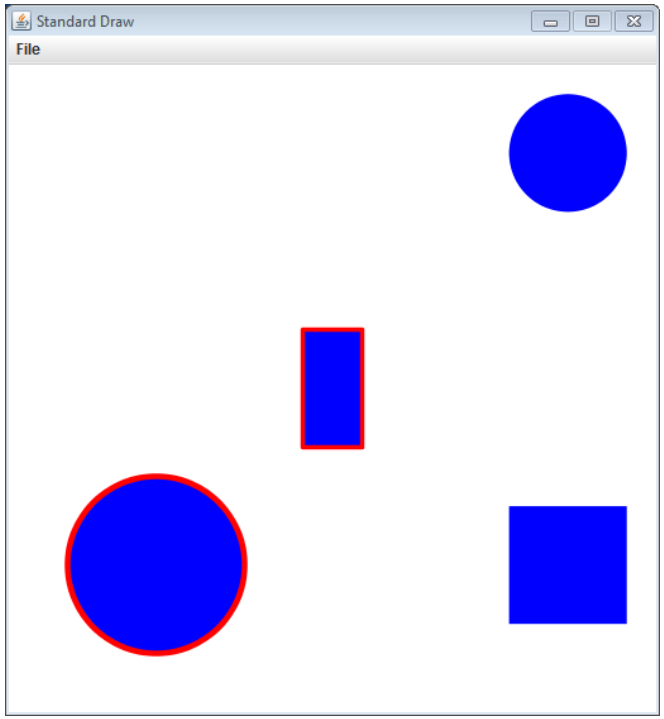


- A shape object hierarchy
  - Classes that "extend"
  - Classes that "implement"

- Java interfaces
  - How Java handles multiple-inheritance problem
    - Avoiding the dreaded deadly diamond of death
  - A class promises to implement a set of methods
    - Implementation left to the class

Shape object hierarchy

- Represent shapes that can:
  - Draw themselves
  - Test for intersection with (x, y) coordinate
  - Change color
  - Support:
    - Circles
    - Rectangles
    - Circles with borders
    - Rectangles with borders



Shape

```
private double x, y;
private Color color;

public double getX();
public double getY();
public Color getColor();
public void setColor(double r, double g, double b);
public double distance(double x, double y);
public abstract void draw();
public abstract boolean intersects(double x, double y);
```

1) Can we create a Shape object?
2) Which are abstract classes?
3) Which are concrete classes?
4) Which are subclasses of Shape?
5) Which are subclasses of Circle?
6) Which methods are overridden?

Circle extends Shape

```
private double radius;

public double getRadius();
public void draw();
public boolean intersects(double x,
                          double y);
```

Rectangle extends Shape

```
private double width, height;

public double getWidth();
public double getHeight();
public void draw();
public boolean intersects(double x,
                          double y);
```

CircleBorder extends Circle

```
private double thickness;
private Color borderColor;

public void draw();
public void setBorderColor(double r,
                           double g,
                           double b);
public void changeBorderThickness(double d);
public void setBorderThickness(double d);
```

RectangleBorder extends Rectangle

```
private double thickness;
private Color borderColor;

public void draw();
public void setBorderColor(double r,
                           double g,
                           double b);
public void changeBorderThickness(double d);
public void setBorderThickness(double d);
```

Border objects

- CircleBorder and RectangleBorder
  - Share two identical instance variables
  - Share three identical methods
  - Can we somehow consolidate?
    - Not really since we want to be related to Shape
    - But shapes like Circle and Rectangle don't want borders

CircleBorder extends Circle

```
private double thickness;
private Color borderColor;

public void draw();
public void setBorderColor(double r,
                           double g,
                           double b);
public void changeBorderThickness(double d);
public void setBorderThickness(double d);
```

RectangleBorder extends Rectangle

```
private double thickness;
private Color borderColor;

public void draw();
public void setBorderColor(double r,
                           double g,
                           double b);
public void changeBorderThickness(double d);
public void setBorderThickness(double d);
```

Interfaces

- Java interfaces
  - Java's alternative to multiple inheritance
  - Classes promise to implement same API
    - An interface is just a list of abstract methods
    - If two classes implement same interface, they can live in the same array for polymorphic goodness

  - Example uses of interfaces:
    - Allow any object type to be easily sorted
    - GUI event listeners
      - e.g. When a button is pushed
    - Classes that can run in their own thread

Bordered interface

```
// Interface for a shape that has a border that can be a
// different color and has a variable pen thickness.
public interface Bordered
{
   public abstract void setBorderColor(double r,
                                       double g,
                                       double b);

   public abstract void setBorderThickness(double thickness);

   public abstract void changeBorderThickness(double delta);
}
```

All methods in an interface are abstract (`abstract` keyword is optional). Implementation is not allowed (except in Java 1.8).

Also no instance variables allowed (except for constants declared `public static final`).

A class adds `implements Bordered` to the class declaration.

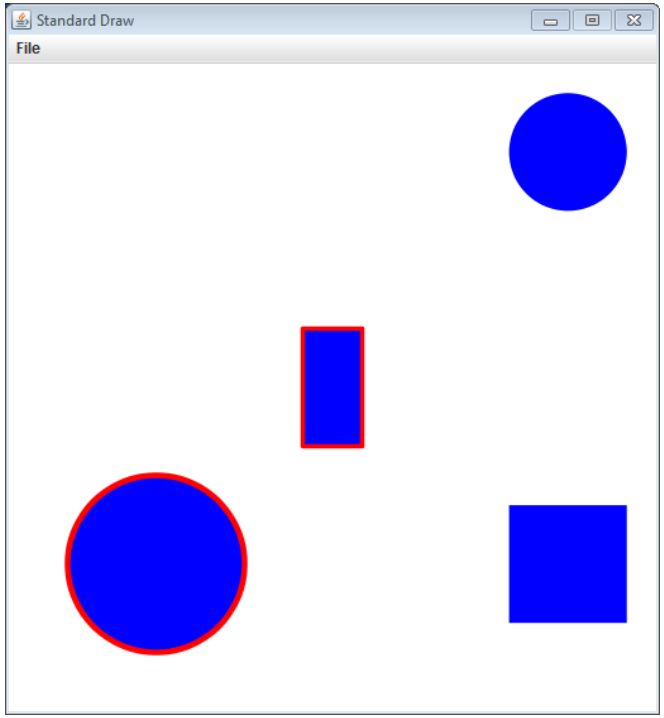
The class must then implement the three methods in interface Bordered.

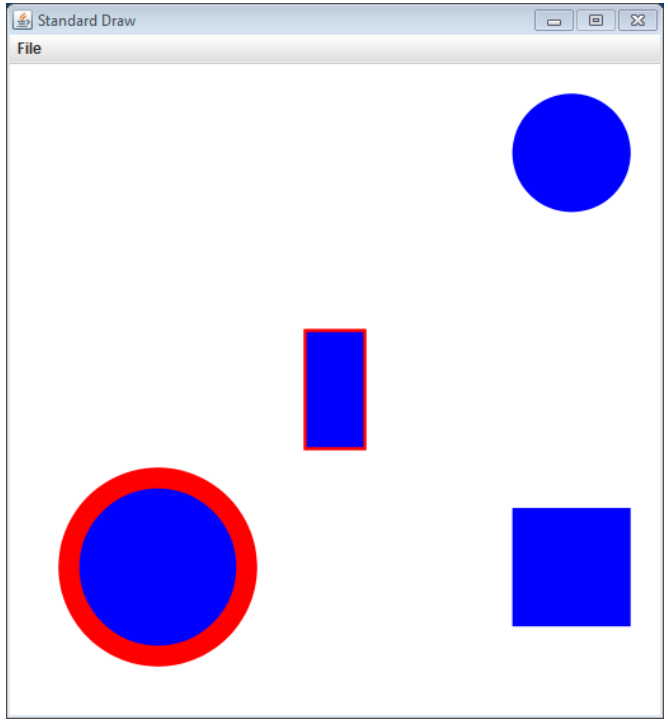
```
public class CircleBorder extends Circle implements Bordered
```

```
public class RectangleBorder extends Rectangle implements Bordered
```

GrowShape

- Show a bunch of Shape objects
  - If mouse is over a shape:
    - Temporarily change object's color
    - If object has a border, permanently grow the border





GrowShape

```
public static void main(String [] args)
{
    Shape [] shapes = new Shape[4];

    shapes[0] = new RectangleBorder(0.5, 0.5, 0.1, 0.2);
    shapes[1] = new CircleBorder(0.2, 0.2, 0.15);
    shapes[2] = new Circle(0.9, 0.9, 0.1);
    shapes[3] = new Rectangle(0.9, 0.2, 0.2, 0.2);

    while (true)
    {
       StdDraw.clear();
       double x = StdDraw.mouseX();
       double y = StdDraw.mouseY();
       for (Shape shape : shapes)
       {
           if (shape.intersects(x, y))
           {
              shape.setColor(0.3, 0.1, 0.5);
              if (shape instanceof Bordered)
                 ((Bordered) shape).changeBorderThickness(0.001);
           }
           else
              shape.setColor(0.0, 0.0, 1.0);

           shape.draw();
       }
       StdDraw.show(100);
    }
}
```

Polymorphic array holding objects in the Shape hierarchy. Some have borders, some don't.

Only increase the border on objects that implements the Bordered interface.

You must check using `instanceof` before casting object.

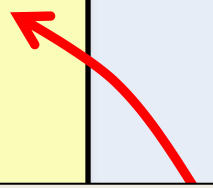
Interface in the real world: Iteration

- Enhanced for-loop
  - Move through all items in many data types
    - e.g. arrays, `ArrayList`, `HashSet`, `Stack`, `LinkedList`
  - Caller doesn't know how items stored
  - Requires data type implement the iterable interface

Examples of iteration with built-in classes

```
String [] namesA = new String[2];
namesA[0] = "Bob";
namesA[1] = "Abe";
for (String s: namesA)
   System.out.println("array: " + s);

ArrayList<String> namesB = new ArrayList<String>();
namesB.add("Bob");
namesB.add("Abe");
for (String s: namesB)
   System.out.println("ArrayList: " + s);

HashSet<String> namesC = new HashSet<String>();
namesC.add("Bob");
namesC.add("Abe");
for (String s: namesC)
   System.out.println("HashSet: " + s);

Stack<String> namesD = new Stack<String>();
namesD.push("Bob");
namesD.push("Abe");
for (String s: namesD)
   System.out.println("Stack: " + s);

LinkedList<String> namesE = new LinkedList<String>();
namesE.add("Bob");
namesE.add("Abe");
for (String s: namesE)
   System.out.println("LinkedList: " + s);
```

```
array: Bob
array: Abe


ArrayList: Bob
ArrayList: Abe


HashSet: Abe
HashSet: Bob


Stack: Bob
Stack: Abe


LinkedList: Bob
LinkedList: Abe
```



The order of iteration through objects may depend on the collection and its implementation.

Looping with iterators

```
ArrayList<String> list = new ArrayList<String>();
list.add("Abe");
list.add("Bob");
list.add("Carol");

for (String s : list)
   System.out.println(s);


Iterator<String> i = list.iterator();
while (i.hasNext())
{
   String s = i.next();
   System.out.println(s);
}
```

Create some iterable collection.

Normal enhanced for-loop version printing out all the elements.

Using iterator object to explicitly loop over all elements in collection.

- To make a collection iterable:
  - Must implement an `iterator()` method that returns an `Iterator` object
  - The `Iterator` class must include two methods:
    - `hasNext()` - Returns a `boolean` if more items
    - `next()` - Returns an item from the collection

Iterators implement an interface

```
public interface Iterator<Item>
{
   boolean hasNext();
   Item next();
   void remove();
}
```

"Removes from the underlying collection the last element returned by the iterator (optional operation)."

-Java API

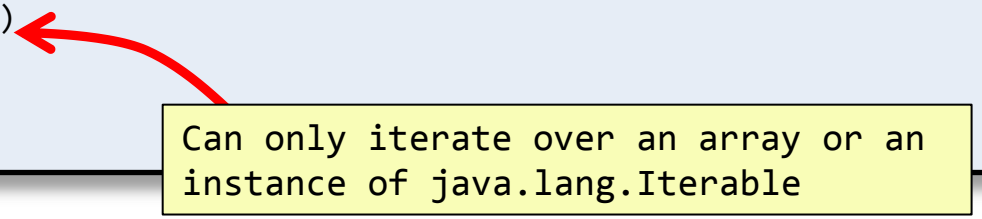
"Interleaving iteration with operations that modify the data structure is best avoided."

-Robert Sedgewick, Kevin Wayne

- Iterable collection returns an instance of a private inner class implementing this interface
  - Object tracks where this iterator is within the collection
  - For a collection backed by an array → integer index
  - For a collection backed by a linked list → a pointer

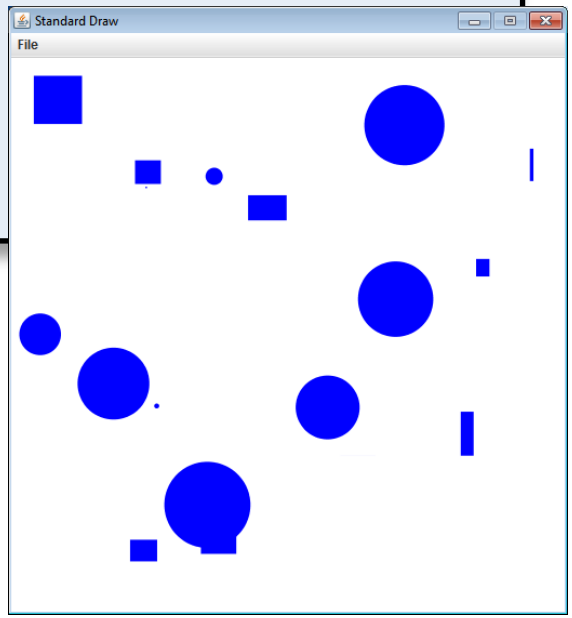
Collection of random shapes

```
public class RandomShapes
{
    private Shape [] shapes;

    public RandomShapes(int num)
    {
        if (num <= 0)
            return;
        shapes = new Shape[num];
        for (int i = 0; i < num; i++)
        {
            double x    = Math.random();
            double y    = Math.random();
            if (Math.random() < 0.5)
                shapes[i] = new Circle(x, y, Math.random() * 0.1);
            else
                shapes[i] = new Rectangle(x, y, Math.random() * 0.1, Math.random() * 0.1);
        }
    }

    public static void main(String [] args)
    {
        RandomShapes shapes = new RandomShapes(20);
        for (Shape s : shapes)
            s.draw();
    }
}
```

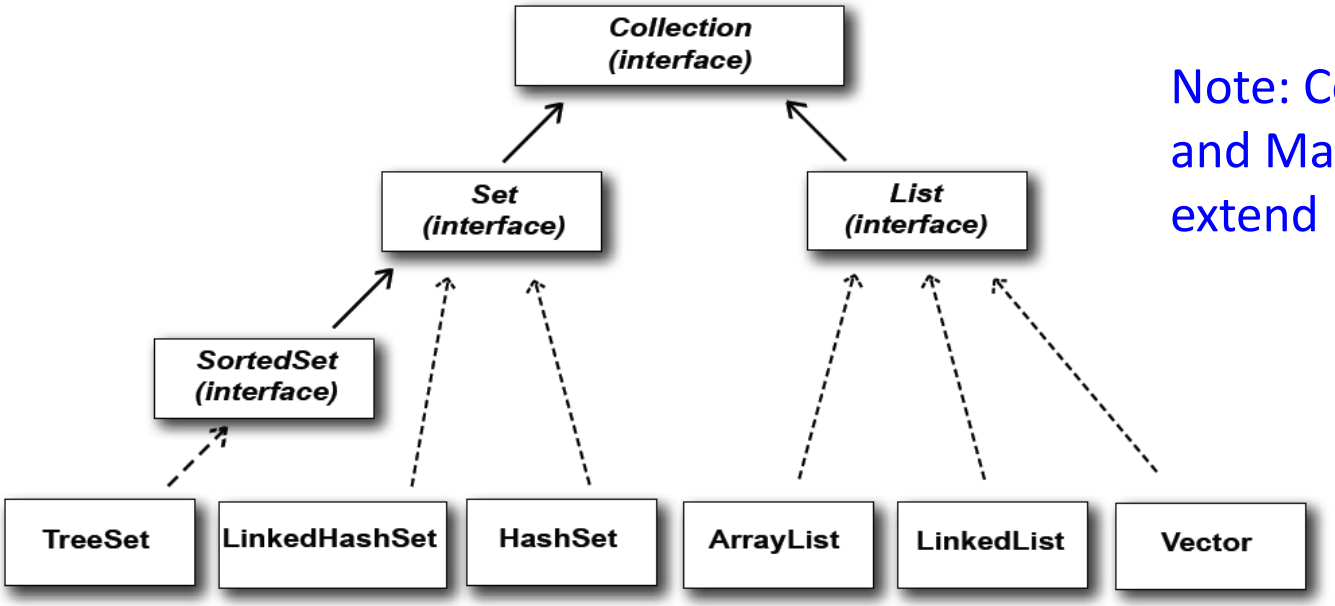


Making random shapes iterable

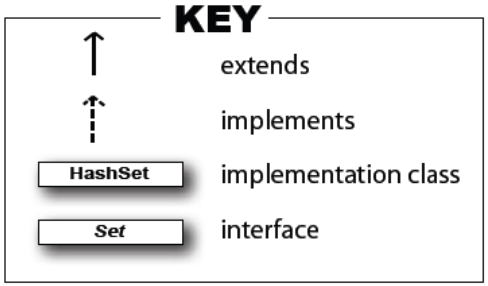
```
import java.util.Iterator;

public class RandomShapes implements Iterable<Shape>
{
    private Shape [] shapes;

    public Iterator<Shape> iterator()
    {
        return new ArrayIterator();
    }

    private class ArrayIterator implements Iterator<Shape>
    {
        private int i = 0;
        public boolean hasNext() { return i < shapes.length; }
        public Shape next()      { return shapes[i++];       }
        public void remove()     {                           }
    }
    ...
}
```

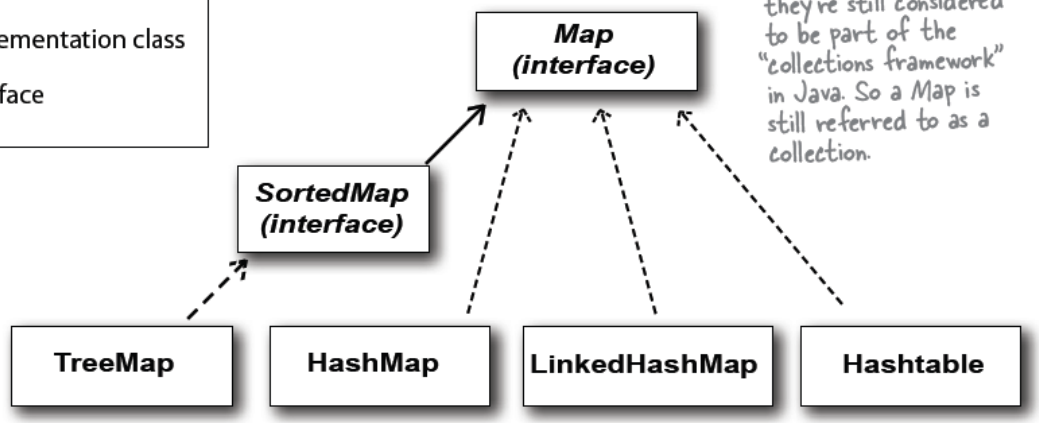


Interfaces in the real world: Collections



Note: Collection and Map also extend Iterable





Maps don't extend from java.util.Collection, but they're still considered to be part of the "collections framework" in Java. So a Map is still referred to as a collection.

Interfaces in the real world: Sorting objects

- Goal: Print `DictEntry`'s, most probable first
  - Read from file
  - Put in file order in a `ArrayList<DictEntry>` object
  - Sort the `ArrayList`
  - Iterate through list, printing each entry

```
and     0.0148
is      0.00987
in      0.00904
to      0.00791
was     0.00676
of      0.00648
for     0.00535
the     0.00535
or      0.00527
that    0.00494
...
```

Sorting a DictEntry list: failure

```
 6    public static void main(String [] args)
 7    {
 8        ArrayList<DictEntry> entries = new ArrayList<DictEntry>();
 9        try
10        {
11            Scanner scan = new Scanner(new File(args[0]));
12            while (scan.hasNext())
13                entries.add(new DictEntry(scan.next(),
14                                          scan.nextDouble()));
15            scan.close();
16        }
17        catch (FileNotFoundException e)
18        {
19            e.printStackTrace();
20            return;
21        }
22        Collections.sort(entries);
23        for (DictEntry e : entries)
24            System.out.println(e);
25    }
```

Bound mismatch: The generic method sort(List<T>) of type Collections is not applicable for the arguments (ArrayList<DictEntry>). The inferred type DictEntry is not a valid substitute for the bounded parameter <T extends Comparable<? super T>>

Making objects sortable

- In order to use `Collection.sort()`:
  - Reference type stored in the `ArrayList` must "`implements Comparable`"
  - i.e. Class must have a `compareTo()` method

Method Summary	
int	compareTo(Object o) Compares this object with the specified object for order.

Method Detail

compareTo

```
public int compareTo(Object o)
```

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

DictEntry in ascending order

```
public class DictEntry implements Comparable<DictEntry>
{

   public int compareTo(DictEntry other)
   {
      if (getProb() > other.getProb())
         return 1;
      else if (getProb() < other.getProb())
         return -1;
      else
         return -0;
   }
...
```


```
% java SortDictEntry entries.txt
moore              5.79E-5
bull               5.79E-5
craft              5.79E-5
periodically       5.79E-5
founder            5.79E-5
nick               5.79E-5
sanctions          5.79E-5
manufactured       5.79E-5
drill              5.8E-5
...
```

DictEntry in descending order

```
public class DictEntry implements Comparable<DictEntry>
{

   public int compareTo(DictEntry other)
   {
      if (getProb() > other.getProb())
         return -1;
      else if (getProb() < other.getProb())
         return 1;
      else
         return 0;
   }
...
```

```
% java SortDictEntry entries.txt
and      0.0148
is       0.00987
in       0.00904
to       0.00791
was      0.00676
of       0.00648
for      0.00535
the      0.00535
or       0.00527
...
```

`this` keyword

- Use 1: Reference hidden instance variables

```
public class Circle
{
   private double x, y, vx, vy, r;

   public Circle(double x, double y, double vx, double vy, double r)
   {
      this.x  = x;
      this.y  = y;
      this.vx = vx;
      this.vy = vy;
      this.r  = r;
   }
}
```

`this` keyword

- Use 2: Call constructor in your own class

```
public class Circle
{
   private double x, y, vx, r;

   public Circle(double x, double y)
   {
      this.x = x;
      this.y = y;
   }

   public Circle(double x, double y, double vx, double vy, double r)
   {
      this(x, y);

      this.vx = vx;
      this.vy = vy;
      this.r  = r;
   }
}
```

`this` keyword

- Use 3: Get a reference to yourself

```
...
public double getDistance(Circle other)
{
   return getDistance(this, other);
}

// Compute the Euclidean distance between the centers of two circles
public static double getDistance(Circle c1, Circle c2)
{
   double deltaX = c1.getX() - c2.getX();
   double deltaY = c1.getY() - c2.getY();

   return Math.sqrt(deltaX * deltaX + deltaY * deltaY);
}
...
```

super keyword

- Use 1: Call a constructor in superclass

```
public abstract class Tile
{
   private double size = 0.0;
   private char   ch   = '\0';

   public Tile(double size, char ch)
   {
      this.size = size;
      this.ch = ch;
   }

   public char getCharacter() { return ch;  }
   public double getSize()    { return size; }

   public abstract void draw(double x, double y);
}
```

```
public class LetterTile extends Tile
{
   public LetterTile(double size)
   {
      super(size, (char) StdRandom.uniform((int) 'A', (int) 'Z' + 1));
   }
   ...
}
```

super keyword

- Use 2: Call a method in super class

```
public abstract class Tile
{
   ...
   public double getSize()    { return size; }
   ...
}
```

```
public class LetterTile extends Tile
{
   ...
   public double getSize()
   {
      final double FUDGE_FACTOR = 1.1;
      return getSize() * FUDGE_FACTOR;
   }

   public static void main(String [] args)
   {
      LetterTile tile = new LetterTile(10.0);
      System.out.println(tile.getSize());
   }
}
```

super keyword

- Use 2: Call a method in super class

```
public abstract class Tile
{
   ...
   public double getSize()    { return size; }
   ...
}
```

```
public class LetterTile extends Tile
{
   ...
   public double getSize()
   {
      final double FUDGE_FACTOR = 1.1;
      return super.getSize() * FUDGE_FACTOR;
   }

   public static void main(String [] args)
   {
      LetterTile tile = new LetterTile(10.0);
      System.out.println(tile.getSize());
   }
}
```

Java 1.8: default methods

- Default methods
  - Implementation in an interface!
  - Concrete class can't implement multiple interfaces with default method with same signature
    - Unless overridden in concrete class, use interface name and `super` to actually use default methods
  - Still no instance variables
    - Except `public static final` constants

```
public interface Dimensions
{
    public double getHeight();
    public double getWidth();
    public double getLength();

    public default double getVolume()
    {
        return getHeight() * getWidth() * getLength();
    }
}
```

Summary

- Namespaces & data encapsulation
  - Access modifiers, package, import
- Object inheritance
  - Share code between similar objects
  - Polymorphism: put objects related by inheritance into a single collection and treat the same
  - Abstract vs. Concrete classes
- Java interfaces - 100% abstract class
  - A promise to implement a set of methods
  - Objects unrelated by inheritance can live together
  - A class can implement multiple interfaces
  - Java 1.8: default interface methods can have implementation
- this and super keywords
  - Call method (including constructors) in your own class (this) or parent class (super)