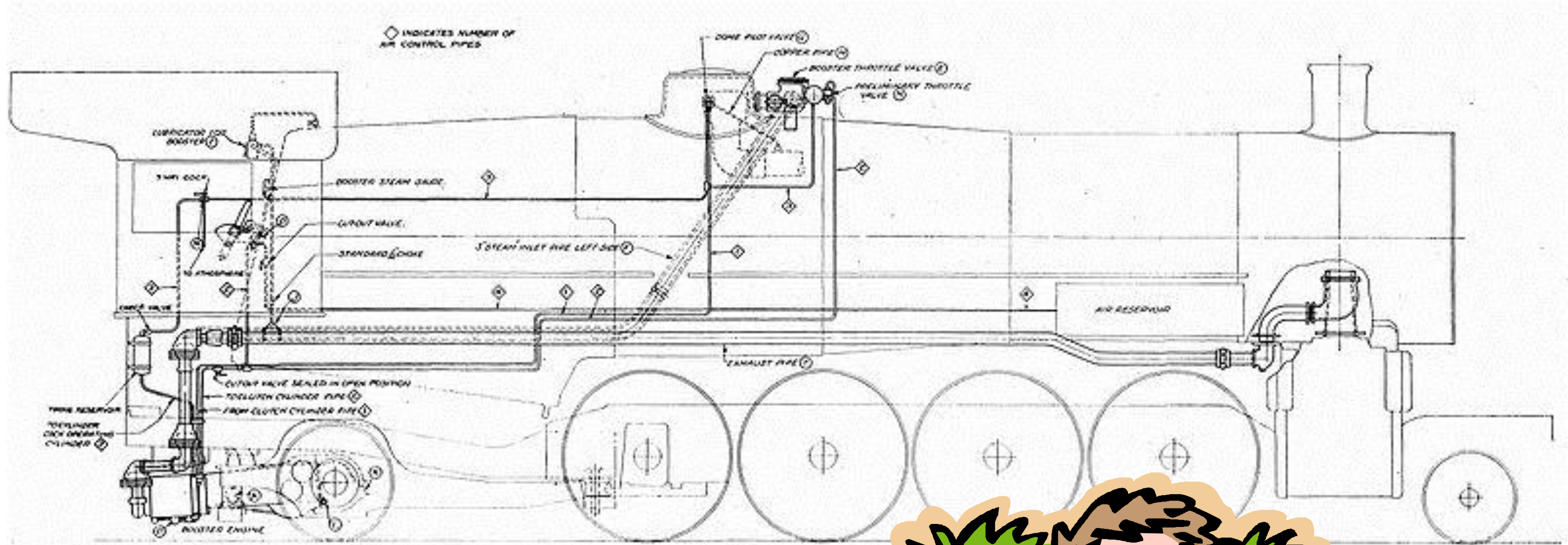
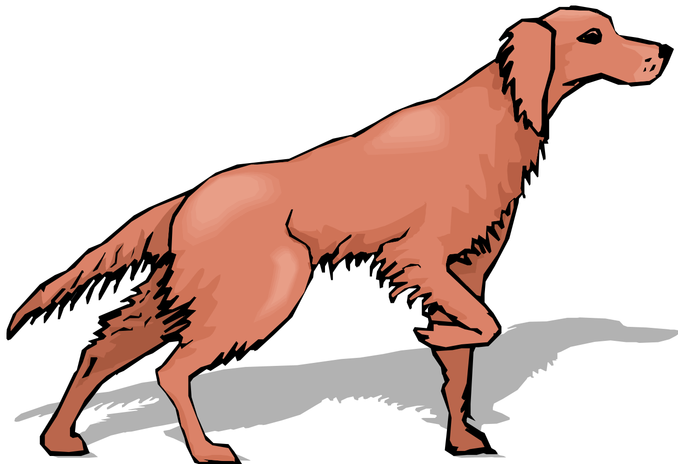
OOP in Java





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

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

Inheritance

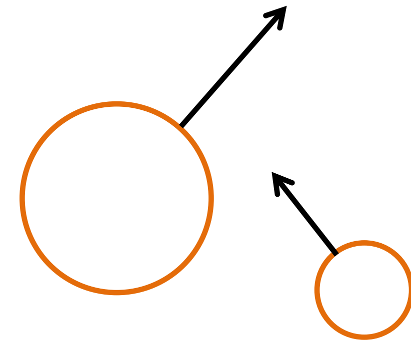
- One class can "extend" another
  - Parent class: shared vars/methods
  - Child class: more specific vars/methods
    - Children extend their parent

- 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

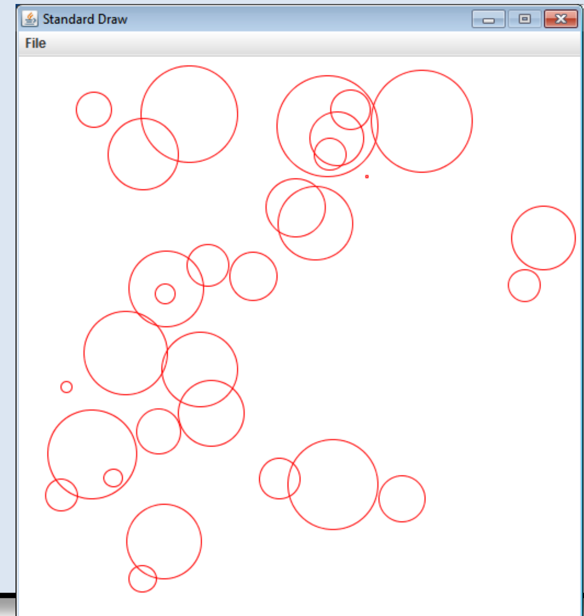


Bouncing circle class

```
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;
   }
}
```

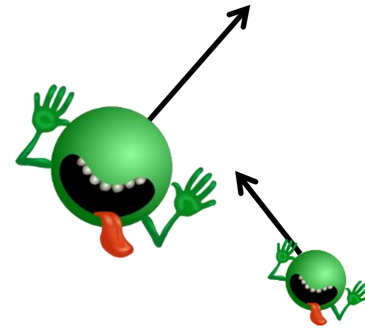
Bouncing circle client

```
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



Bouncing circular image class

```
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()
   { ... }
}
```

All this code appeared in the Circle class!

Bouncing circular image class

```
public class CircleImage extends Circle
{
   protected String image;

   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(x, y, image, r * 2, r * 2);
   }
}
```

This class is a child of the `Circle` class

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

We use protected access modifier so children (even outside our package) can see our instance variables

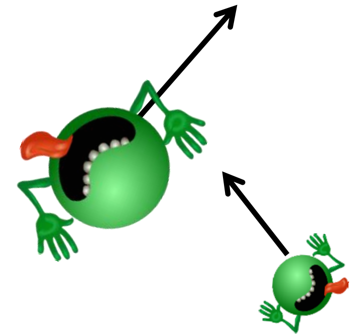
Overridden version of `draw()` method, this one draws a picture scaled according to the radius.

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
{
   protected int angle;

   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(x, y, image, r * 2, r * 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:

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

  - Loop through them separately:

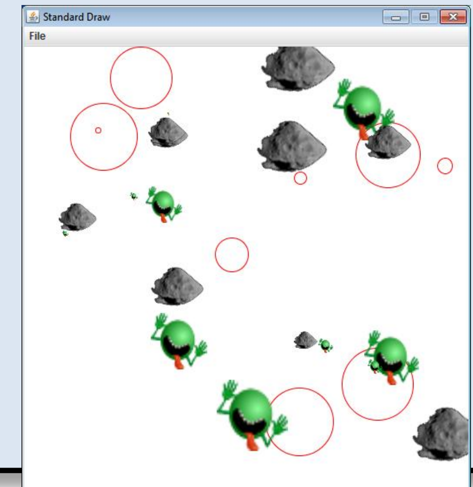
```
for (int i = 0; i < circles1.length; i++)
   circles1[i].updatePos();

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);
}
```

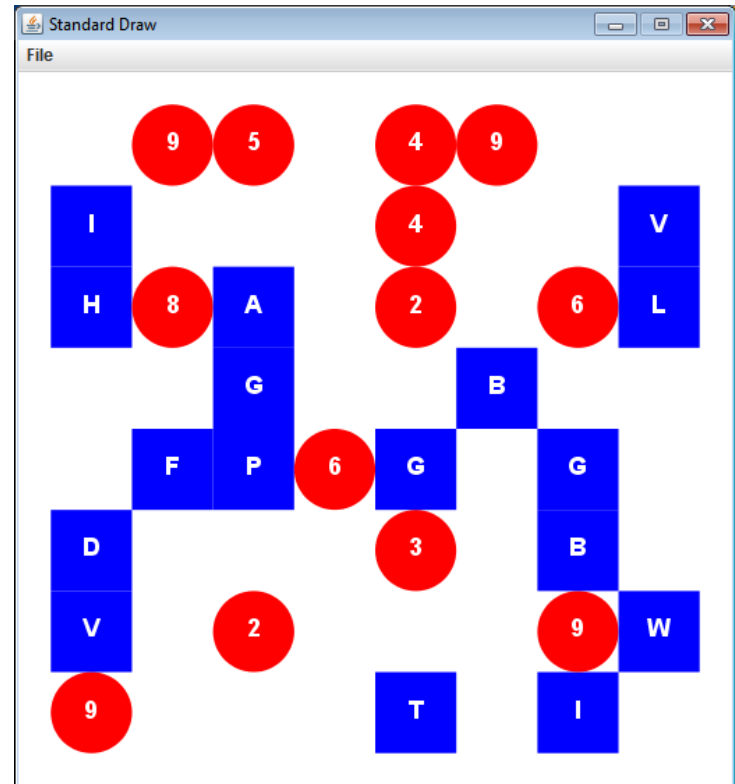
circles[i] may be a Circle, CircleImage or CircleImageRotate object

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

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

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

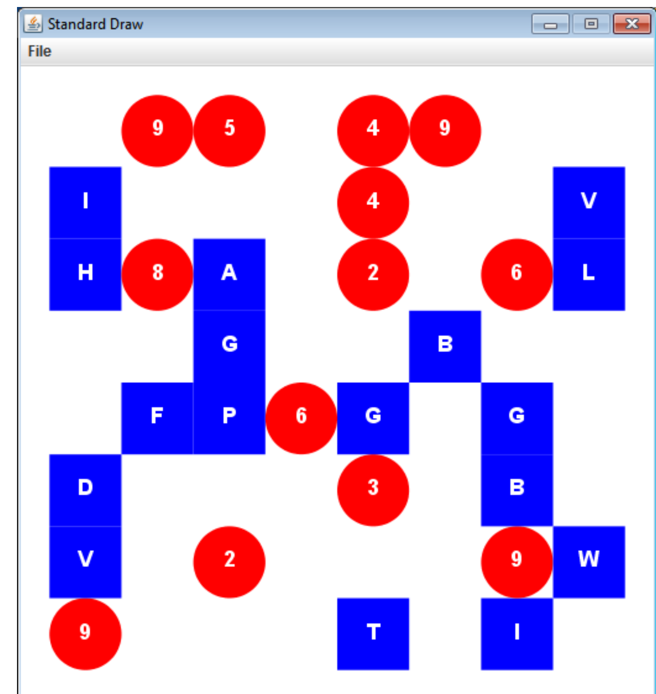


Designing the Tile game

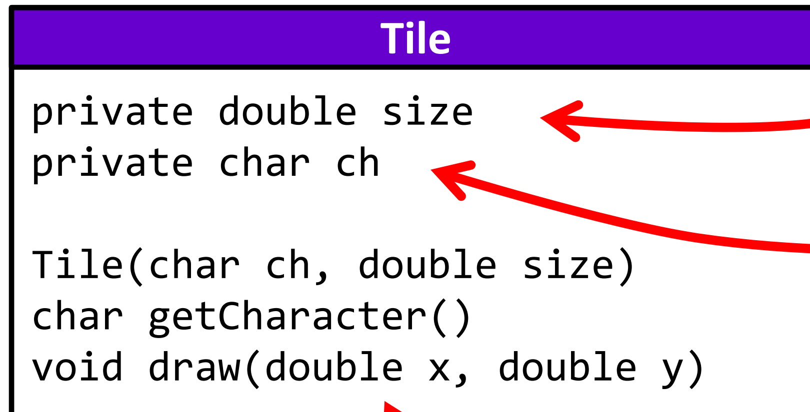
- Use a 2D array for the N x N grid
  - Array element `null` if no tile there
  - Otherwise reference to letter/number tile object
  - Start by randomly placing non-overlapping tiles

```
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?



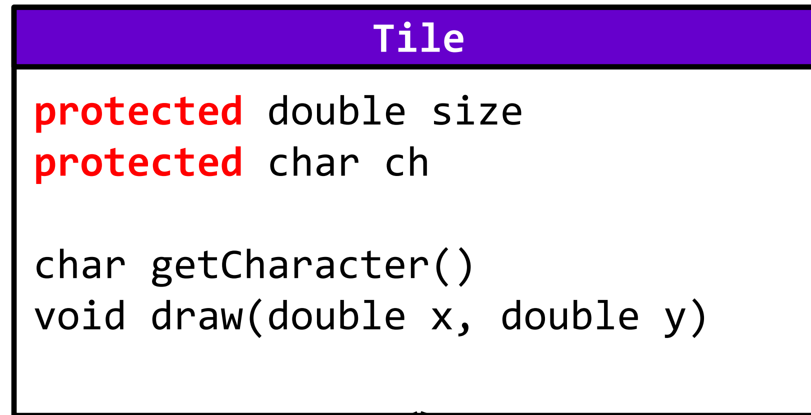
How big to draw ourselves (a Tile object doesn't know the number of grid cells or screen canvas size).

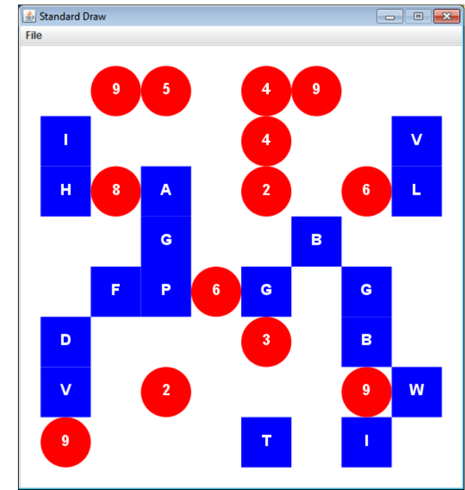
The character appearing on this Tile.

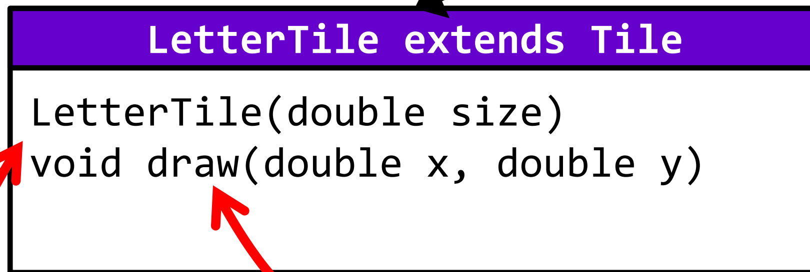
Draw ourselves, somebody tells us our center (x,y) location.

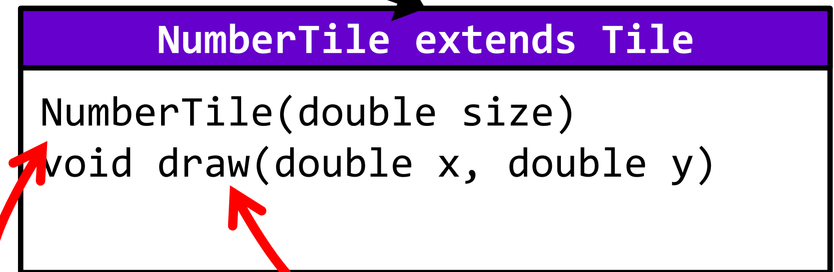
- Problem: `draw()` has to check character to know how to draw itself
  - Any method whose behavior depends on tile type needs similar conditional logic

Tile class hierarchy









Draw a square tile

Draw a circle tile

Construct using a random letter A-Z

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 this method for free

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

   public char getCharacter()
   {
      return ch;
   }

   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)
   {
      this.ch = (char) StdRandom.uniform((int) 'A',
                                          (int) 'Z' + 1);
      this.size = size;
   }

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

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

Randomly assign a letter between A and Z

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

Concrete NumberTile class

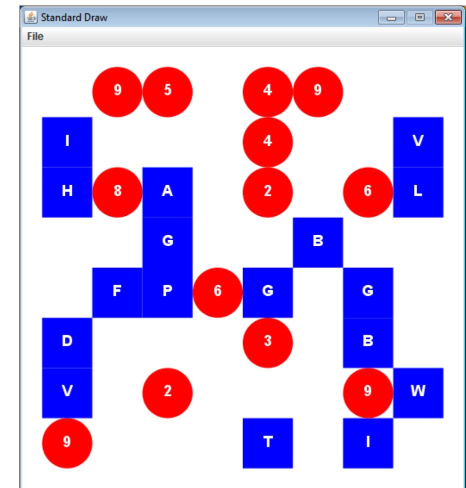
```
public class NumberTile extends Tile
{
    public NumberTile(double size)
    {
        this.ch = (char) StdRandom.uniform((int) '0',
                                          (int) '9' + 1);
        this.size = size;
    }

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

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

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

```
Tile [][] tiles // 2D array storing LetterTile and NumberTile objs
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[numTiles][numTiles];
   size = canvasSize / gridSize;

   int added = 0;
   while ((added < numTiles) && (added < gridSize * gridSize))
   {
      int x = (int) (Math.random() * gridSize);
      int y = (int) (Math.random() * gridSize);

      if (tiles[x][y] == null)
      {
         if (Math.random() < 0.5)
            tiles[x][y] = new LetterTile(size);
         else
            tiles[x][y] = new NumberTile(size);
         added++;
      }
   }
}
```

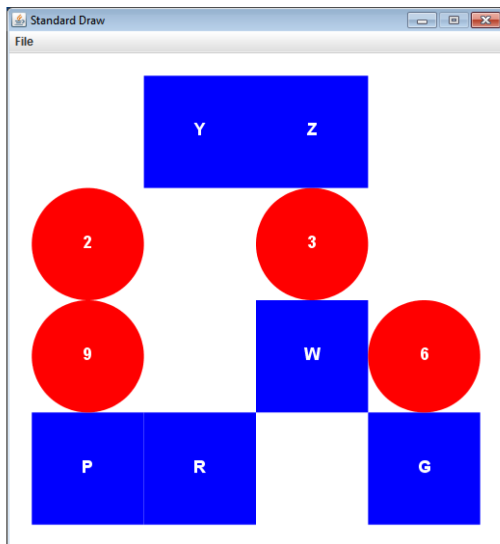
TileBoard drawing

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

    for (int x = 0; x < tiles.length; x++)
    {
       for (int y = 0; y < tiles[x].length; y++)
       {
          if (tiles[x][y] != null)
             tiles[x][y].draw(size * (x + 0.5),
                              size * (y + 0.5));
       }
    }
}
```

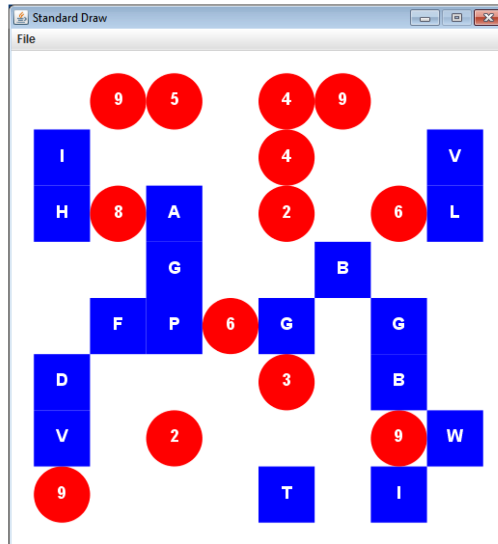
TileGame main program

```java
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();
   }
}
```

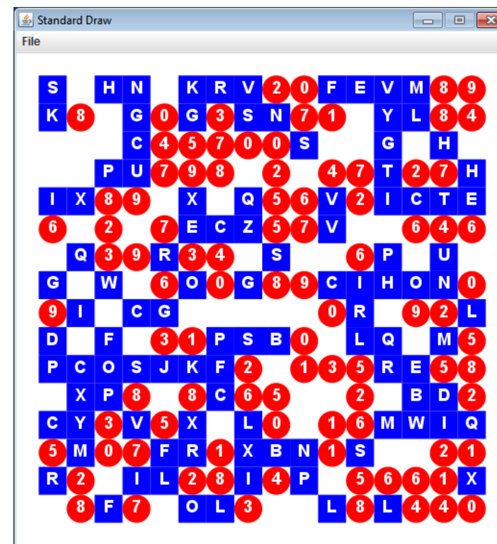
```
% java TileGame 10 4
```




```
% java TileGame 30 8
```




```
% java TileGame 200 16
```

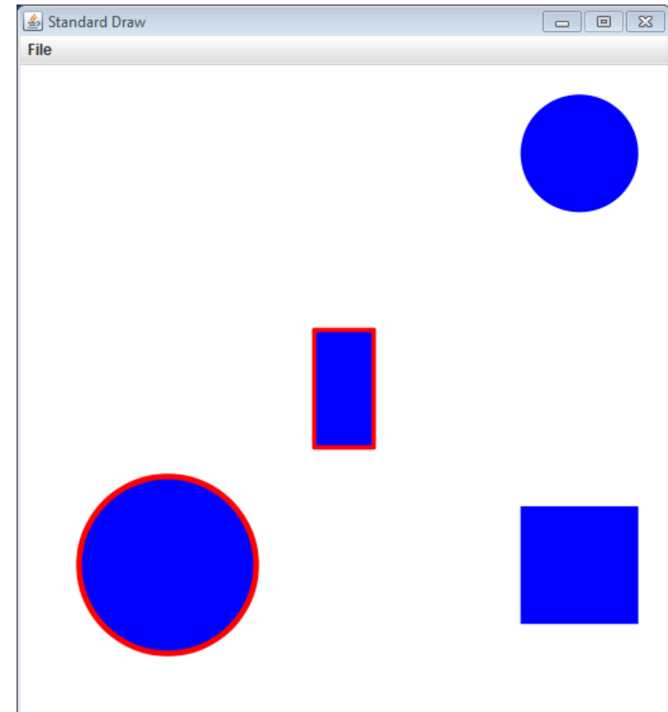


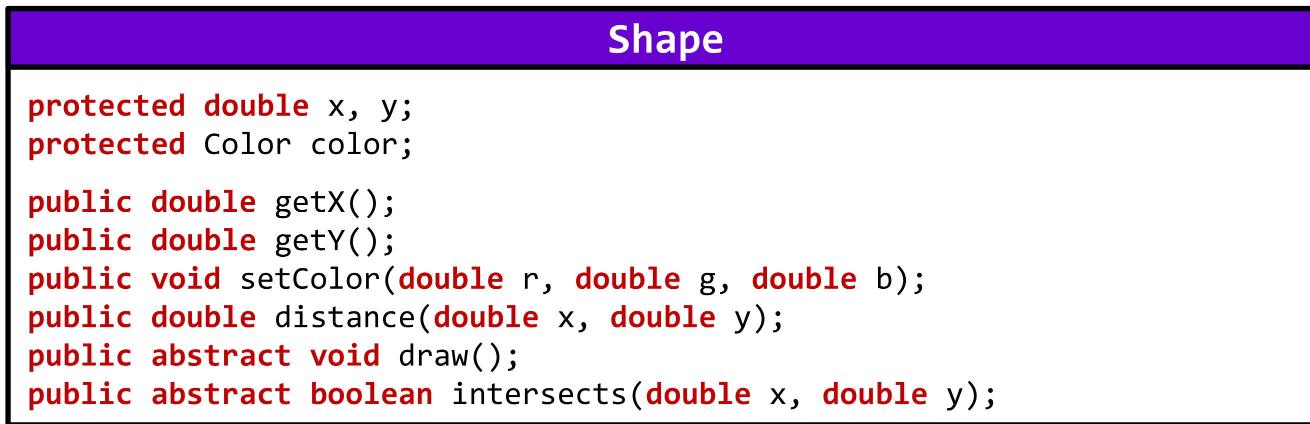
Interfaces

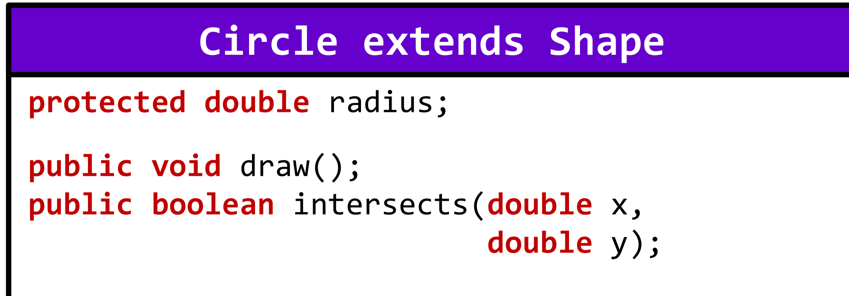
- A shape object hierarchy
  - Classes that "extend"
  - Classes that "implement"
- Java interfaces
  - How Java handles multiple-inheritance problem
  - A class promises to implement a set of methods
    - Implementation left to the class

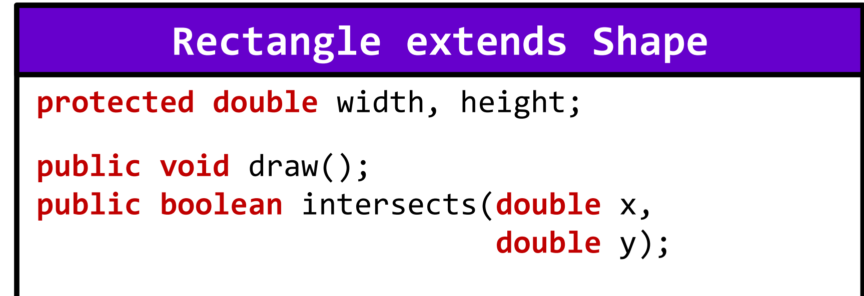
Shape object hierarchy

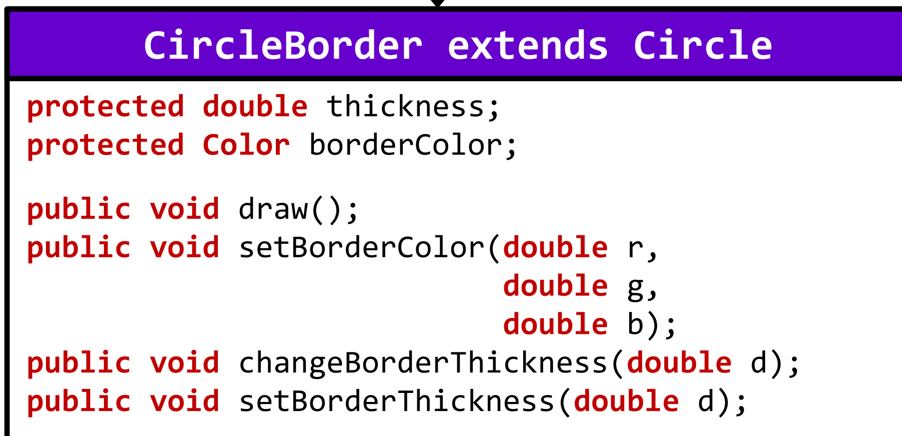
- Represent shapes that can:
  - Draw themselves
  - Test for intersection with (x, y) coordinate
  - Change color
  - Support:
    - Circles
    - Rectangles
    - Circles with borders
    - Rectangles with borders

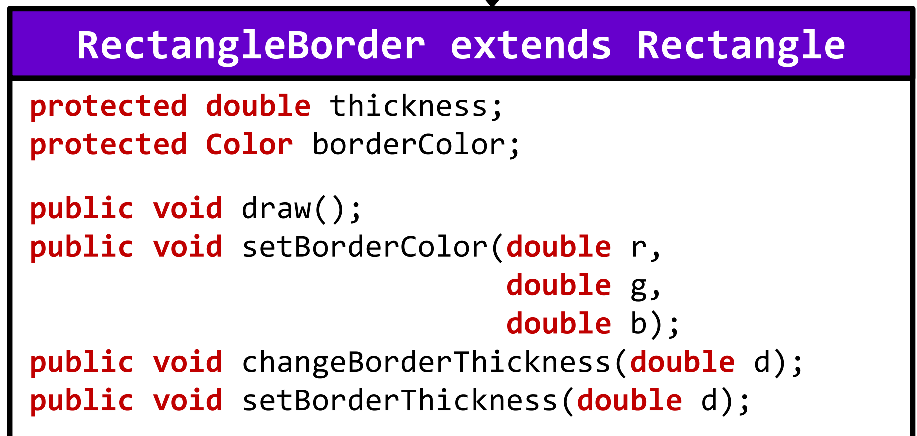



Shape
protected double x, y;
protected Color color;
public double getX();
public double getY();
public void setColor(double r, double g, double b);
public double distance(double x, double y);
public abstract void draw();
public abstract boolean intersects(double x, double y);


Circle extends Shape
protected double radius;
public void draw();
public boolean intersects(double x,
double y);


Rectangle extends Shape
protected double width, height;
public void draw();
public boolean intersects(double x,
double y);


CircleBorder extends Circle
protected double thickness;
protected 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
protected double thickness;
protected Color borderColor;
public void draw();
public void setBorderColor(double r,
double g,
double b);
public void changeBorderThickness(double d);
public void setBorderThickness(double d);

Shape

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

public double getX();
public double getY();
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?
7) Can we change radius to private?

Circle extends Shape

```
protected double radius;

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

Rectangle extends Shape

```
protected double width, height;

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

CircleBorder extends Circle

```
protected double thickness;
protected 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

```
protected double thickness;
protected 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

```
protected double thickness;
protected 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

```
protected double thickness;
protected 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 declared in an interface must be abstract. Implementation is not allowed!

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

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

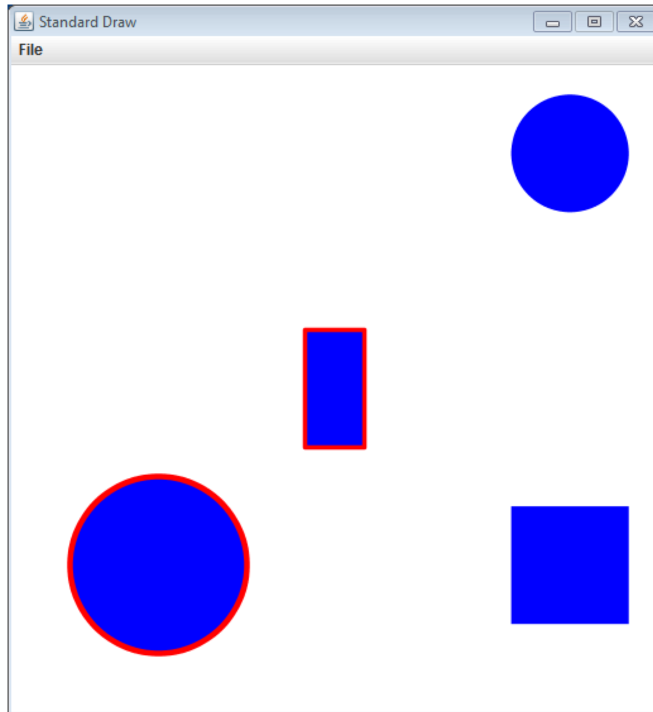
The class must then implement the three methods in interface Bordered.

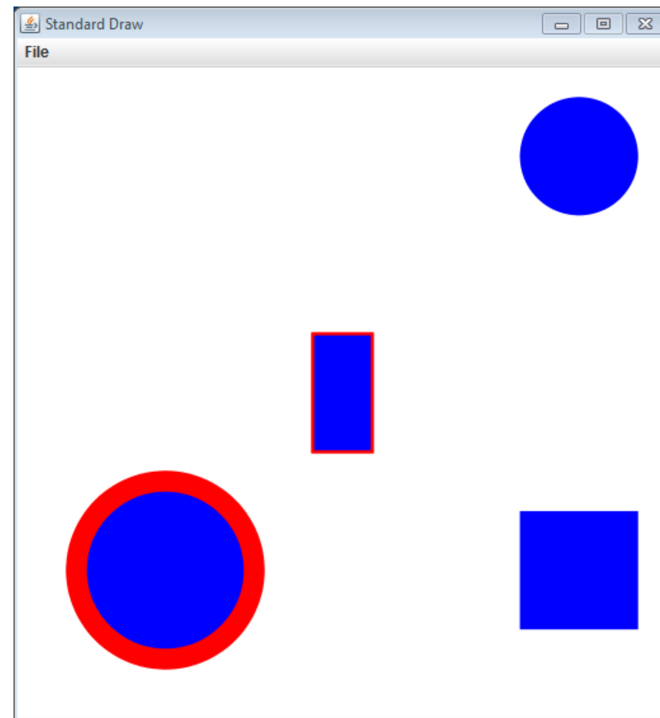
```
public class CircleBorder extends Circle implements Bordered
```

```
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

```java
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 implement the Bordered interface.

You must check using instanceof before attempting to cast object.

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