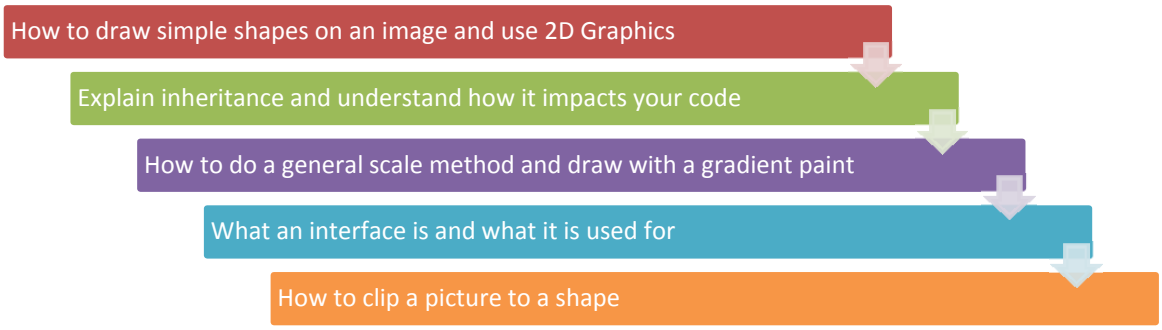
Module IV: Inheritance and Interfaces

Learning Objectives



Bitmapped versus Vector Graphics

The pixel-by-pixel modifications to images that we have been exploring so far are sometimes called bitmapped, or Raster graphics. Raster graphics draw by changing the color of pixels, one at a time.

Because many pictures are made up of well-defined primitives like lines and circles, some applications use vector graphics to represent images. These pictures are essentially represented using programs that can produce the picture. For example, Postscript, Flash, and AutoCAD all use vector graphics. Not only are vector graphics images smaller to store, but they also have the advantage that they are easy to change and can be scaled up (that is to say, when running the program to draw the image, the size, or scale of the image can be specified as a parameter).

The pictures we have drawn so far have required a fair amount of set up for very simple tasks. For example, to draw a circle, we need to define the coordinates, set the color, and so on. Yet the methods we were using can't do things like change the thickness of the circle's border line, apply a texture to a line or area, or use a gradient fill. For example, to create a gradient fill for a rectangle with the Graphics class, we would need to draw a stack of filled rectangles starting from the lightest one at the bottom right and the darkest one at the top left:

```
/**
 * draw a series of rectangles to create a gradient fill
 */
public void drawFilledRectangles() {
  Graphics graphicsObj = this.getGraphics();
  Color color = null;

  //loop 25 times
  for (int i = 25; i > 0; i--)  {
    color = new Color(i * 10, i * 5, i);
    graphicsObj.setColor(color);
    graphicsObj.fillRect(0, 0, i * 10, i * 10);
  }
}
```

As always in programming, the solution to a situation where common tasks are difficult to do is to encapsulate those tasks in objects and methods so that the solutions can be easily used and re-used. In this case, the tasks are so common that they are included in the Java library. The result is a more object-oriented approach to drawing. For example, instead of calling drawOval() or fillOval() you create a Ellipse2D object and ask a 2d Graphics object to draw or fill it.

Geometric shapes such as Ellipse2D are in the java.awt.geom package, which you can import using import java.awt.geom.*; In addition, Java provides a Graphics2D object. Java's 2D graphics infrastructure has support for different types of brushes, line thickness, dashed lines, cubic curves and general paths, gradients and textures. It can also move, rotate, scale, and shear text and graphics. It can also create composite images. Java provides a demo of these features that you can explore yourself. You can find source code for it online. By default, it should be included with the java installation on your hard drive, and will be located at "*/path/to*/jdk/demo/jfc/Java2D/"

Introduction to Java 2D

To use Java 2D, you'll need to do the following:

```
// Import (you could use import java.awt.*; instead)
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Color;
import java.awt.BasicStroke;
import java.awt.geom.*;

//Create a picture
Picture pictureObj = new Picture(800,600);

//Get the graphics object from your picture
Graphics gObj = pictureObj.getGraphics();

//Cast the Graphics class to Graphics2D
Graphics2D g2 = (Graphics2D) gObj;
```

Once you have a Graphics2D object, there are many things you can do with it. Graphics2D and the related classes in java.awt.geom.* add more advanced drawing abilities. Many details are documented on the Java pages for Graphics2D and java.awt.geom. Here are some examples:

Example: Drawing a line with Graphics2D

```
//Set up the stroke if desired (type of pen) pass in the width (a float)
g2.setStroke(new BasicStroke(2));

//Set up any Color, GradientPaint, or TexturePaint
g2.setPaint(Color.BLUE);

// these are commented out because you would need  to create the TexturePaint or GradientPaint
// objects first (look at the documentation mentioned above to see how)
//g2.setPaint(blueToPurpleGradient);
//g2.setPaint(texture);
//Create a line
Line2D line2D = new Line2D.Double(0.0,0.0,100.0,100.0);

//Draw it
g2.draw(line2D);
// You could use a geometric shape instead, by creating an object such as a Rectangle2D and
// then drawing (or filling) it as in the code  below g2.fill(rectangle2D);
```

This is the first time I've left you the job of exploring something in documentation instead of explaining it. As you expand your coding skills, you will find documentation, online examples, and other external resources become more and more critical to your progress. To help you along, here are the docsfor java.awt.geom.Rectangle2D, the docs for java.awt.GradientPaint and the docs for java.awt.TexturePaint

Let's revisit our program for drawing an X on an image. Previously, we used graphicsObj.drawLine() to draw two lines on the picture. Let's say we want to pass in a line width and color. Using Graphics2D, we easily change the width of the line, using g2.setStroke(new BasicStroke(width)). We will also need to create a Line2D object for each of the lines before completing the drawing. Test it with:

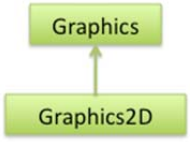
```
Picture pictureObj = new Picture(400,300);
pictureObj.drawFancyX(Color.RED,4);
pictureObj.show();
```

This is what the code would look like:

```
/**
 * Method to draw an X on an image from corner to corner
 * @param color the color the X should be
 * @param strokeWidth the width the lines should be
 */
public void drawFancyX(Color color, float strokeWidth) {
  //get the graphics2D object
  Graphics2D g2 = (Graphics2D) this.getGraphics();

  // set the color to draw the X with
  g2.setPaint(color);

  //set the stroke width
  g2.setStroke(new BasicStroke(strokeWidth));

  // create the lines
  Line2D firstLine = new Line2D.Double(0.0, 0.0, (double)this.getWidth(), (double)this.getHeight());
  Line2D secondLine = new Line2D.Double(0.0, (double)this.getHeight(), (double)this.getWidth(), 0.0);

  // draw the lines
  g2.draw(firstLine);
  g2.draw(secondLine);
}
```

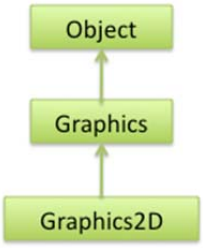
Inheritance

Now that you've played with Graphics2D for a while, let's talk a little bit about what it is. Graphics2D is a kind of Graphics class, but it adds new methods and properties. For this reason, we say say that Graphics2D *inherits* from Graphics. Since Graphics2D inherits from Graphics, you can use it to do all the same things you can do in Graphics.



Graphics2D inherits from Graphics (which inherits from Object)

How do we know that **Graphics2D** inherits from **Graphics**? The Java API documentation will tell you what the parent of a class is. For example, if you click "java.awt" in the main window of the documentation and then scroll down to "Graphics2D," you will see a brief description mentioning that it *extends* Graphics (meaning, it inherits from Graphics). If you click on "Graphics2D", you will see that Graphics2D inherits from Graphics, which in turn inherits from Object. In the Java programming language, each class can have one parent, and each parent can have an unlimited number of children. An inheritance hierarchy shows all the parents and children of a class.



The parents of Graphics2D

Remember that we said that you can use **Graphics2D** to do all the same things you can do in Graphics? What this means, specifically, is that you can call any of the methods implemented by Graphics when you have an object of type Graphics2D. Just

as a child inherits characteristics of their parents, a class in java will inherit characteristics of its parent class. Specifically, it inherits fields and methods. The Java API documentation can help you figure out what methods a class has inherited. If you click on "java.awt" and "Graphics2D" again, scroll down, and you will see a section titled "Methods inherited from class java.awt.Graphics" An example is the **drawLine()** method. It belongs to (is implemented in) the **Graphics** class, but it can be called on a **Graphics2D** class. If you have **Graphics2D graphics2D = (Graphics2D) pic.getGraphics(),** you can call **graphics2D.drawLine(x1, x2, y1, y2)** just like you could if **graphics2D** was of type **Graphics**.

Example:

Although we didn't mention this earlier for pedagogical reasons, Graphics can be used to copy an image, as follows:

```
/**
 * Copy from a source picture into this one
 * @param source the source image
 * @param x the X location to start the copy from
 * @param y the Y position to start the copy from
 */
public void copy(Picture source, int x, int y) {
  //get the graphics object
  Graphics g = this.getGraphics();
  // copy the image
  g.drawImage(source.getImage(), x, y, null);
}
```

Since Graphics2D inherits from Graphics, we can use the exact same method with a Graphics2D object, as follows:

```
/**
 * Copy from a source picture into this one
 * @param source the source image
 * @param x the X location to start the copy from
 * @param y the Y position to start the copy from
 */
public void copyG2D(Picture source, int x, int y) {
  Graphics g = this.getGraphics();
  Graphics2D g2D = (Graphics2D)g;
  // copy the image
  g2D.drawImage(source.getImage(), x, y, null);
}
```

You can see that both methods work the same way using the following code:

```
Picture p1 = new Picture("img1.jpg");
Picture p2 = new Picture("img2.jpg");
p1.copy(p2,194,304);
p1.show();
p1.copyG2D(p2,194,304);
p1. show ();
```

learn by doing

The Java API documentation will be a very important resource for you moving forward. No one an remember every detail of all of the classes provided in the Java API. Instead, good programmers k now to make use of documentation (this is one reason it's so very important to carefully document your code -- you won't even remember everything your own code does once you write enough of it!)

A. Let's explore the inheritance hierarchy of some of the classes we've used so far. Find the Color class in the API. What is it's parent class? _____
B. How many subclasses (child classes) does Color have? _____
C. How many subclasses (child classes) does Graphics have? _____
D. Can you find "GeneralPath"? What can you find out about it from looking at the documentation?

solution

A. Object. Color is in the Java.awt package.
B. 2 - Look under the text "Direct Known Subclasses": they are "ColorUIResource" and "SystemColor".
C. 2 - Look under the text "Direct Known Subclasses": they are "Graphics2D" and "DebugGraphics"
D. Look in "java.awt.geom". As you gain more familiarity with Java, you will learn more and more about classes from their documentation. Some things to note: GeneralPath is a type of two dimensional path (since it inherits from Path2D). It "represents a geometric path constructed from straight lines, and quadratic and cubic (Bézier) curves. It can contain multiple subpaths." Sounds like a pretty powerful class for describing shapes!

General Scaling

. What if we want to scale to a specified size? Or what if we want to scale up in x and down in y? We can use the class AffineTransform in the java.awt.geom package. Affine transform is used as follows:

- Create an object of the class AffineTransform
- Set up the scaling using the method scale()
- Use the AffineTransform object when you draw the image

```
/**
 * Scales an image by a specified x factor and y factor
 * @param xFactor the factor to scale the X axis by
 * @param yFactor the factor to scale the Y axis by
 */
public Picture scale(double xFactor, double yFactor) {
  // set up the scale transform
  AffineTransform scaleTransform = new AffineTransform();
  scaleTransform.scale(xFactor, yFactor);

  // create a new picture object that is the right size
  Picture result = new Picture ((int) (getWidth() * xFactor), (int) (getHeight() * yFactor));

  //get the graphics object to draw on the result
  Graphics g = result.getGraphics();
  Graphics2D g2 = (Graphics2D) g;

  // draw the current image onto the result image scaled
  g2.drawImage(this.getImage(), scaleTransform, null);
  return result;
}
```

Note... Look closely at the method declaration above. This method is different from most of the others we've defined. Instead of public void scale(..., it is written as public Picture scale(.... That means that the method *returns* an object of type Picture instead of modifying the existing picture.

How do we make this happen? We create a *new*, blank picture whose size is defined by the scaling factor (see line 13). We get the Graphics2D object associated with the new picture (see line 16). We then call drawImage() on the new images Graphics2D object passing in the image associated with this (the current picture), and the AffineTransform scaleTransform (see line 20). Once we've done that, the new picture will contain a scaled down version of the current picture. All that is left is to *return* the new picture, which we do by calling the special keyword return (see line 21).

In the past, we have always called image manipulation methods on the picture we wish to modify. If we write a method that *returns* a modified picture instead, we give much more control to the person who uses our method: They can save the returned picture, store it in place of the original, or do anything else they want to it.

The general scale method creates and returns a new picture of the appropriate size. So to see the result we need to save a reference to it in a variable

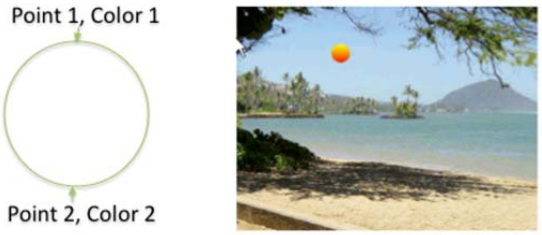
```
Picture p = new Picture("img.jpg");
Picture p1 = p.scale(2.0,0.5);
p1.show();
```

Gradients

We gave an example of stacked rectangles drawn using a loop. Taken together, they create the impression of a *gradient* fill.

This same effect can be created by filling a single rectangle with a paint that changes from one color to another, java.awt.GradientPaint. This is done as follows:

- Specifying a point and the color at that point
- Then specify a second point and the color at that point
- Create a GradientPaint using those points, and paint with it. There will be a change from one color to the other.



A gradient is created by specifying two colors at two points. The image at right shows a sun with a gradient from yellow (top of circle) to red (bottom of circle).

```
public void drawGradientSun(int x, int y, int width, int height) {
  Graphics g = this.getGraphics();
  Graphics2D g2 = (Graphics2D) g;

  // create the gradient for painting from yellow to red with yellow at the top of the sun and red at the bottom
  float xMid = (float) (width / 0.5 + x);
  GradientPaint gPaint = new GradientPaint(xMid, y, Color.yellow, xMid, y + height, Color.red);

  // set the gradient and draw the ellipse
  g2.setPaint(gPaint);
  g2.fill(new Ellipse2D.Double(x, y, width, height));
}
```

You can test the code as follows

```
Picture p = new Picture("img.jpg");
p.drawGradientSun(201,80,40,40);
p.show();
```

Interfaces

Something interesting went on in the example I just gave. Look at the description of java.awt.GradientPaint. You should see that it inherits from Object, and implements two interfaces (Paint and Transparency).

But if you look at the documentation for java.awt.Graphics2D, and look at the setPaint() method, you should see that it takes a parameter of type Paint. How can we pass this method a java.awt.Color object or a java.awt.GradientPaint object? They are not related (one does not inherit from the other), so they aren't of the same class type.

Look closely at the API documentation for both classes, and you'll see that both of these classes implement the Paint interface.

> Note...Why does Java have interfaces? One reason is that they allow classes to have more complex relationships than simple inheritance does. A class may only inherit from one parent (for example, GradientPaint inherits from Object). However, it may *implement* many interfaces (for example, GradientPaint implements Paint and Transparency.

In the past, we've declared methods that take parameters whose type is defined as a primitive (like int) or a class (like Picture). Now we can add interfaces (like Paint) to the list of things that can be used to specify the type of a parameter (or variable, for that matter). When you assign a variable or call a method that has an interface as its type, any object whose class implements that interface *or* inherits from a class that implements the interface can be used.

interface

> (Definition) An interface takes its name from a concept that you are familiar with. For example, consider a USB interface. It lets you plug in different devices (Camera, disk drive, key drive, *etc.*). The computer doesn't care what the device is, just that it uses the USB interface. Java interfaces are the same. They let you plug in different classes as long as they implement the interface. In Java, an interface is a sort of specification for what messages the class must respond to. Stated differently, an interface is just a list of methods the class must implement. To implement an interface, the class must have implementations of *all* the methods specified in the interface.

Let's compare interfaces and inheritance and primitive types

Inheritance	**Interface**	**Primitive Types**
Lots of classes can inherit from the same object	Lots of classes can implement the same interface	Primitive types are not classes
The syntax to inherit from a class is extends classname	The syntax to implement an interface is implements interface [extends classname]	Primitive types cannot be extended
A class includes fields and methods	An interface includes only methods	A primitive contains only a value
A class can only inherit from a SINGLE object	A class can implement any number of different interfaces	
A variable var can be defined using a class Classname as its type	A variable var can be defined using an interface Interfacename as its type	A variable var can be defined using a primitive as its type
Any instance of Classname can be assigned to var	Any instance of *any* class that implements Interfacename can be assigned to var	Anything value of that primitive type can be assigned to var
var doesn't know what interfaces Classname implements, only what it's class is.	var doesn't know about methods or fields associated with the class implementing Interfacename, only the methods defined in Interfacename	varcontains a value, not a class
If var's type (Classname) implements an interface Interfacename, and var2 is of type Interfacename, then var2 = var is a valid statement. This is because Java knows that Classname implements Interfacename	If var's type is Interfacename; var contains an instance of Classname; and var2 is of type Classname, then var2 = var is *not* a valid statement. Instead, you must use var2 = (Classname) var. This is because Java has no way of knowing which of the classes that implement Interfacename are contained in var	If var and var2 are of the same primitive type, then var2 = var is a valid statement. Otherwise you must use casting to assign var to var2
Inheritance is single, not multiple	Interfaces support "multiple inheritance"	primitives do not support inheritance

Clipping

Clipping an image means removing parts of it to match a pre-defined shape. You can specify a shape to clip the image to when you draw it, and only the portion of the image that is inside that shape will be drawn. For example, this code will clip an image to an ellipse:

```
Ellipse2D.Double ellipse = new Ellipse2D.Double(0,0,width,height);
g2.setClip(ellipse);
g2.drawImage(this.getImage(),0,0,width,height,null);
g2.show();
```

learn by doing

Write a method Picture clipToEllipse() in your Picture class to clip an image to an ellipse. You should design this method to return a new picture just like the scale()method discussed earlier. Testing it: This method will create a new picture and returns it so in order to see if we will need to save a reference to it

```
Picture p = new Picture("<image>");
Picture p2 = p.clipToEclopse();
p2.show();
```

learn by doing

```
public Picture clipToEllipse() {
   int width = this.getWidth();
   int height = this.getHeight();
   Picture result = new Picture(width,height);
   Graphics g = result.getGraphics();
   Graphics2D g2 = (Graphics2D)g;
   Ellipse2D.Double ellipse = new Ellipse2D.Double(0,0,width,height);
   g2.setClip(ellipse);
   g2.drawImage(this.getImage(),0,0, width,height, null);
   return result;
}
```

You can use the java.awt.geom.GeneralPath class to create an arbitrary shape to clip to. You create a new GeneralPath by passing in a Line2D.Double object like so:

GeneralPath myShape = new GeneralPath(new Line2D.Double(startX, startY, endX, endY);

You can then append additional lines to it (to construct a shape), by using the append() method, like so:

myShape.append(new Line2D.Double(newStartX, newStartY, newEndX, newEndY), true)

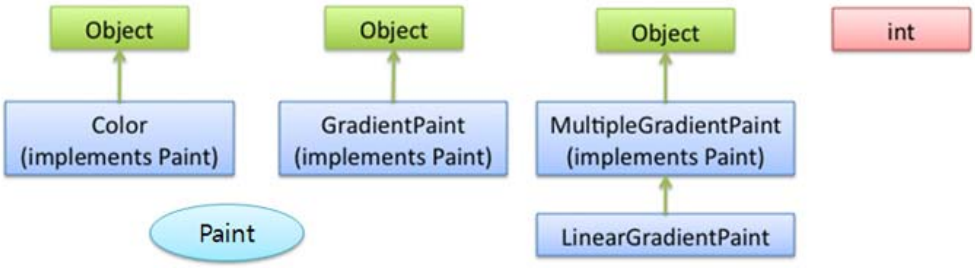
The **true** that is passed as the second parameter tells Java to connect the new line to the previously added line, thus making a continuous path. Once your shape is complete, you can use it as a clipping region for a picture by passing it into **setClip().**

Summary

- You can use the original Graphics class for simple 2D drawing, such as Lines, rectangles, ovals, polygons, strings, images
- You can use the Graphics2D class for more advanced drawing, such as width of paint brush, and geometric objects
- Graphics2D inherits from Graphics. This means it provides the same methods, but also adds others.
- Inheritance means that the child class gets all the fields and methods of the parent class (See this in the API for Graphics2D).
- You can use a class to do general scaling and rotation: java.awt.geom.AffineTransform.
- You can draw with a gradient paint. That changes from one color to another.
- Objects of classes that implement an interface can be said to be of that type, and can be passed as parameters to methods that take that type.
- You can clip a picture to a geometric object, or a general path
- You can create a new picture when you want to make a modification, instead of changing the current one. To do this, you need to specify the *return type* of your method as Picture

We also discussed interfaces, which allow many different classes to share a common set of abilities. Interfaces can be used anywhere types like classes and primitives can be used.

To summarize, here are all the different sorts parameters we can use in a method.



This picture shows several things that could be used as method parameters.

A primitive:

A parameter can be a primitive type such as **int** (red box)

An object whose type is a Class:

A parameter can have the type of a class. For example, a method could be defined that takes type **Object** as a parameter (**public void myMethod(Object myObj)**. In that case, any of the classes shown in the image (blue and green rectangles) would be valid parameters. This is because they all inherit from Object or from something that inherits from Object. Similarly, a method that takes an object of type **MultipleGradientPaint** could also be passed an object of type **LinearGradientPaint** because **LinearGradientPaint** inherits from **MultipleGradientPaint**.

An object whose type is an Interface:

All of the blue boxes are classes that implement **Paint** (an interface) either explicitly (**Color, GradientPaint, MultipleGradientPaint**), or through inheritance (**LinearGradientPaint**). An object created from any of the four classes shown in blue could be passed into a method requiring a parameter of type Paint.