Think Java

CHAPTER 10: MUTABLE OBJECTS

DR. ERIC CHOU

IEEE SENIOR MEMBER

LECTURE 1

Objects

Objects

- As we learned in the previous chapter, an object is a collection of data that provides a set of methods. For example, a String is a collection of characters that provides methods like **charAt** and **substring**.
- Java is an "**object-oriented**" language, which means that it uses objects to represent data and provide methods related to them. This way of organizing programs is a powerful design concept, and we will introduce it a little at a time throughout the remainder of the book.
- In this chapter, we introduce two new types of objects: **Point** and **Rectangle**. We show how to write methods that take objects as parameters and produce objects as return values. We also take a look at the source code for the Java library.

LECTURE 2

Point Objects

Point objects

- The **java.awt** package provides a class named Point intended to represent the coordinates of a location in a Cartesian plane. In mathematical notation, points are often written in parentheses with a comma separating the coordinates. For example, (0,0) indicates the origin, and (x,y) indicates the point x units to the right and y units up from the origin.
- In order to use the Point class, you have to import it:

```
import java.awt.Point;
```

Point objects

- Then, to create a new point, you have to use the new operator:

```
Point blank;
blank = new Point(3, 4);
```

- The first line declares that blank has type Point. The second line creates the new Point with the given arguments as coordinates.
- The result of the new operator is a reference to the new object. So blank contains a reference to the new Point object. Figure 10.1 shows the result.

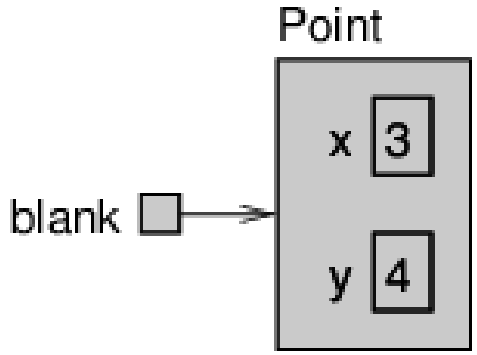


Figure 10.1: State diagram showing a variable that refers to a Point object.

Attributes

- Variables that belong to an object are usually called attributes, but you might also see them called “fields”. To access an attribute of an object, Java uses dot notation. For example:

```
int x = blank.x;
```

- The expression **blank.x** means “go to the object blank refers to, and get the value of the attribute x.” In this case, we assign that value to a local variable named x. There is no conflict between the local variable named x and the attribute named x.
- The purpose of dot notation is to identify which variable you are referring to unambiguously.

Attributes

- You can use dot notation as part of an expression. For example:

```
System.out.println(blank.x + ", " + blank.y);
int sum = blank.x * blank.x + blank.y * blank.y;
```

- The first line displays 3, 4; the second line calculates the value 25.

LECTURE 3

Objects as Parameters

Objects as parameters

- You can pass objects as parameters in the usual way. For example:

```
public static void printPoint(Point p) {
    System.out.println("(" + p.x + ", " + p.y + ")");
}
```

- This method takes a point as an argument and displays its attributes in parentheses. If you invoke **printPoint(blank)**, it displays (3, 4).
- But we don't really need a method like **printPoint**, because if you invoke **System.out.println(blank)** you get:

```
java.awt.Point[x=3,y=4]
```

toString() Method

- **Point** objects provide a method called **toString** that returns a string representation of a point. When you call println with objects, it automatically calls **toString** and displays the result. In this case, it shows the name of the type (**java.awt.Point**) and the names and values of the attributes.
- As another example, we can rewrite the distance method from Section 6.2 so that it takes two Points as parameters instead of four doubles.

```
public static double distance(Point p1, Point p2){
  int dx = p2.x - p1.x;
  int dy = p2.y - p1.y;
  return Math.sqrt(dx * dx + dy * dy);
}
```

- Passing objects as parameters makes the source code more readable and less error-prone, because related values are bundled together.

LECTURE 4

Objects as Return Types

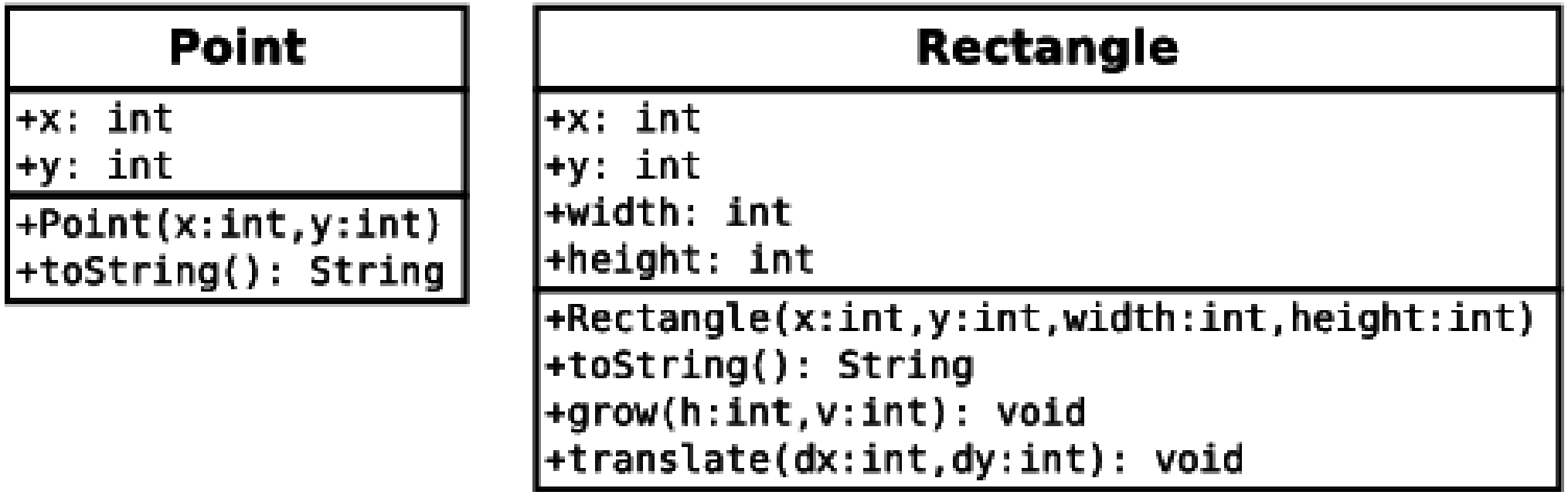
Objects as Return Types

- The java.awt package also provides a class called Rectangle. To use it, you have to import it:

```
import java.awt.Rectangle;
```

- Rectangle objects are similar to points, but they have four attributes: x, y, width, and height. The following example creates a Rectangle object and makes the variable box refer to it:

```
Rectangle box = new
Rectangle(0, 0, 100, 200);
```


Point
+x: int
+y: int
+Point(x:int,y:int)
+toString(): String
Rectangle
+x: int
+y: int
+width: int
+height: int
+Rectangle(x:int,y:int,width:int,height:int)
+toString(): String
+grow(h:int,v:int): void
+translate(dx:int,dy:int): void

Objects as Return Types

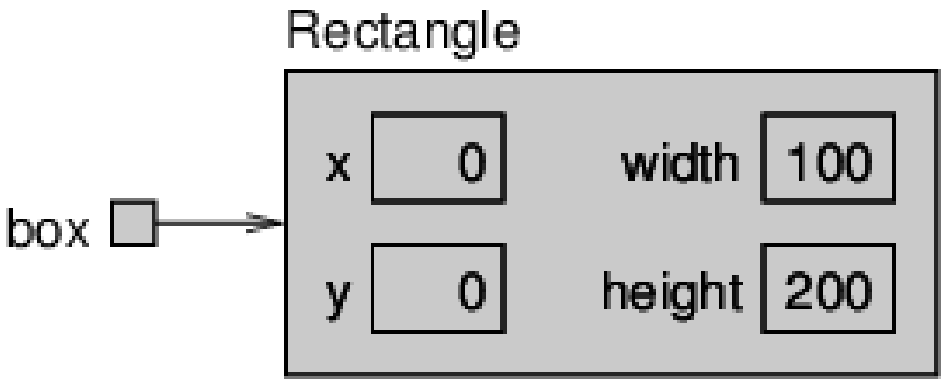


Figure 10.2: State diagram showing a Rectangle object

- If you run System.out.println(box), you get:

```
java.awt.Rectangle[x=0,y=0,width=100,height=200]
```

- Again, **println** uses the toString method provided by **Rectangle**, which knows how to display **Rectangle** objects.

The return type of this method is Point. The last line creates a new Point object and returns a reference to it.

Find Center

- You can write methods that return objects. For example, **findCenter** takes a Rectangle as an argument and returns a Point with the coordinates of the center of the rectangle:

```
public static Point
findCenter(Rectangle box) {
   int x = box.x + box.width / 2;
   int y = box.y + box.height / 2;
   return new Point(x, y);
}
```

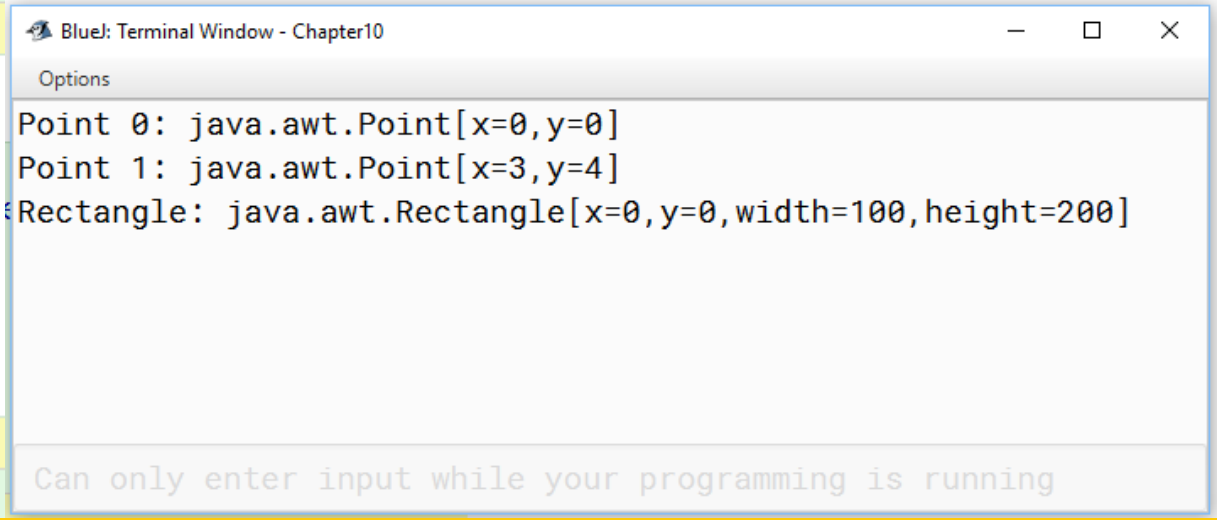
- The return type of this method is Point. The last line creates a new Point object and returns a reference to it.

Demo Program: DataModel0.java

Go BlueJ!!!

DataModel0.java

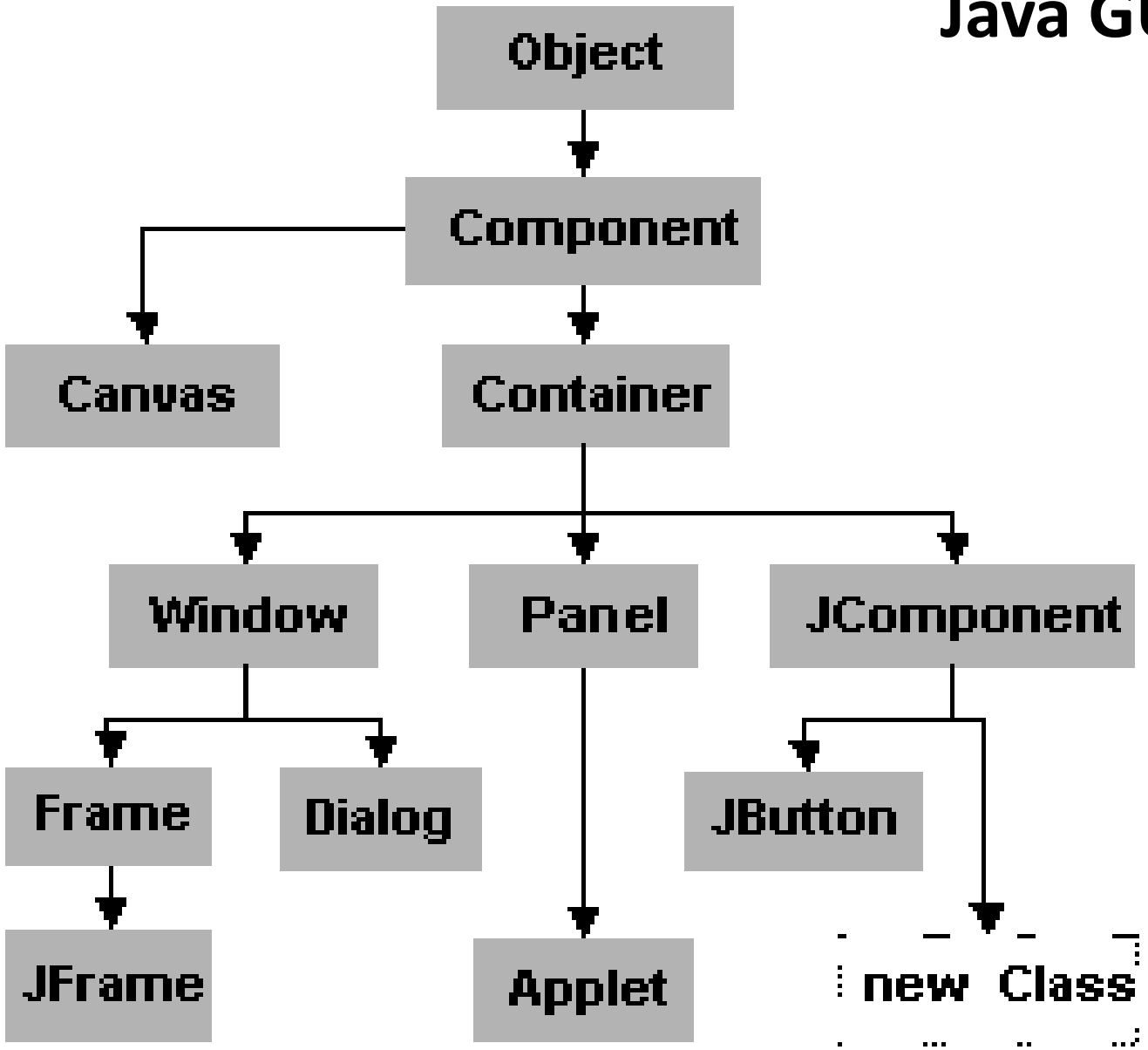
```java
import java.awt.Rectangle;
import java.awt.Point;

/**
 * Write a description of class DataModel0 here.
 *
 * @author (Eric Y. Chou)
 * @version (12/11/2017)
 */
public class DataModel0
{
    public static void main(String[] args){
      Point p1 = new Point(3, 4);
      Point p0 = new Point(0, 0);
      Rectangle box = new Rectangle(0, 0, 100, 200);
      System.out.println("Point 0: "+p0);
      System.out.println("Point 1: "+p1);
      System.out.println("Rectangle: "+box);

     }
}
```



LECTURE 5

Java 2D graphics Appendix B

Java GUI (AWT/Swing)



Java 2D Graphics

AWT and Swing API

- The Java library includes a simple package for drawing 2D graphics, called **java.awt**. **AWT** stands for “Abstract Window Toolkit”.
- **Swing API**, a much more comprehensive set of graphics libraries that enhances the AWT, was introduced as part of Java Foundation Classes (JFC). JFC consists of Swing, Java2D, Accessibility, Internationalization, and Pluggable Look-and-Feel Support APIs.
- We are only going to scratch the surface of graphics programming; you can read more about it in the Java tutorials at https://docs.oracle.com/javase/tutorial/2d/.

Creating graphics

- There are several ways to create graphics in Java; the simplest way is to use **java.awt.Canvas** and **java.awt.Graphics**.
- A Canvas is a blank rectangular area of the screen onto which the application can draw.
- The Graphics class provides basic drawing methods such as **drawLine**, **drawRect**, and **drawString**.

Basic Graphics Terminology

- Frame: Top level container for graphics objects. (JFrame is a frame of Swing version)
- Canvas: drawing pad for Java AWT/Swing which can be the main object (or objects) in a frame.
- paint() is canvas's main drawing function.

Drawing Class

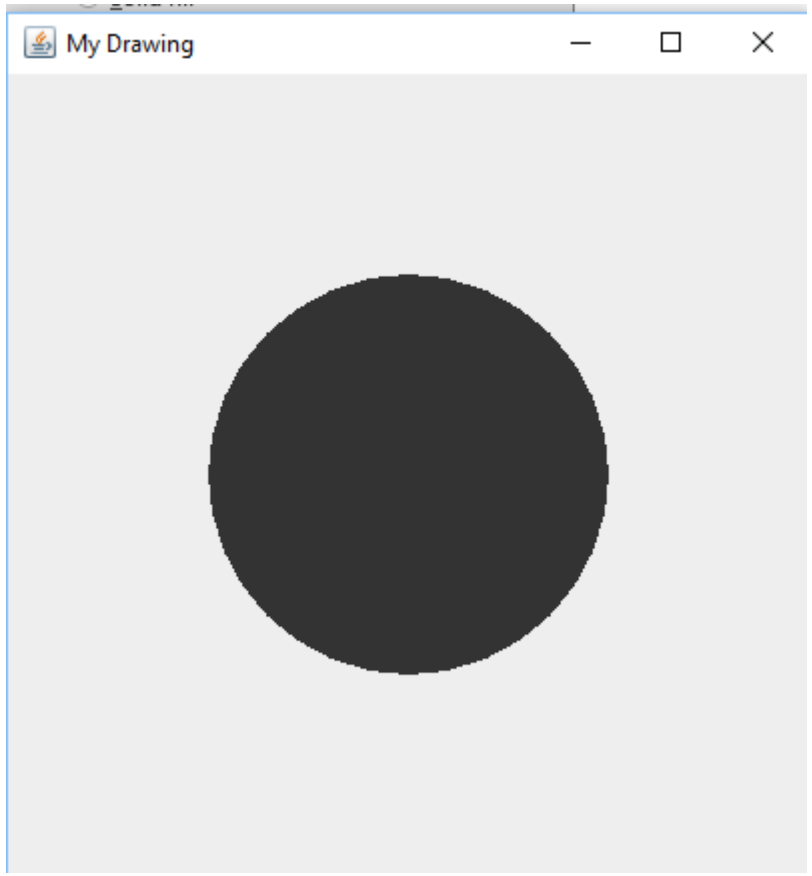
- The **Drawing** class extends Canvas, so it has all the methods provided by **Canvas**, including **setSize**.
- In the main method, we:
  - Create a **JFrame** object, which is the window that will contain the canvas.
  - Create a **Drawing** object (which is the canvas), set its width and height, and add it to the frame.
  - **Pack** the frame (resize it) to fit the canvas, and display it on the screen.
- Once the frame is visible, the **paint** method is called whenever the canvas needs to be drawn; for example, when the window is moved or resized. The application doesn't end after the main method returns; instead, it waits for the **JFrame** to close. If you run this code, you should see a black circle on a gray background.

Drawing Class is a Canvas in A Frame

```
public class Drawing extends Canvas{

    public void paint(Graphics g) {
        g.fillOval(100, 100, 200, 200);
    }

    public static void main(String[] args) {
        JFrame frame = new JFrame("My Drawing");
        Canvas canvas = new Drawing();
        canvas.setSize(400, 400);
        frame.add(canvas);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Demo Program: Drawing.java

Go BlueJ !!!



LECTURE 6

Graphics Methods Appendix C

Java Graphical Coordinates

- You are probably used to Cartesian coordinates, where x and y values can be positive or negative. In contrast, Java uses a coordinate system where the origin is in the upper-left corner. That way, x and y are always positive integers. **Figure B.1** shows these coordinate systems.
- Graphical coordinates are measured in pixels; each pixel corresponds to a dot on the screen.

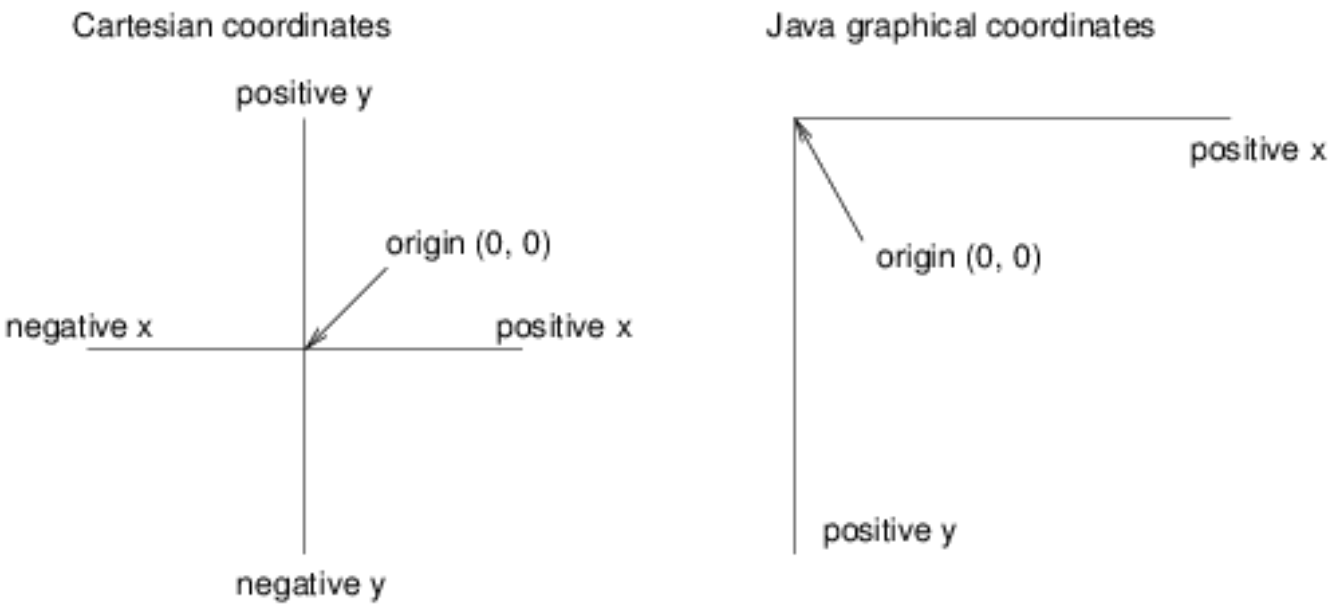


Figure B.1: Diagram of the difference between Cartesian coordinates and Java graphical coordinates.

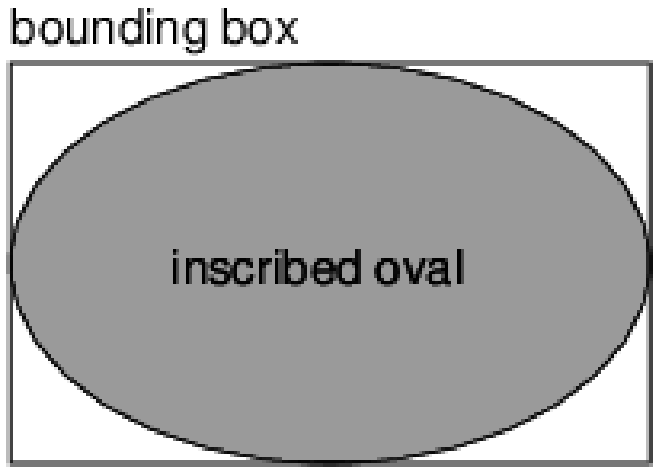
fillOval

- To draw on the canvas, you invoke methods on a Graphics object. You don’t have to create the Graphics object; it gets created when you create the Canvas, and it gets passed as an argument to paint.
- The previous example used **fillOval**, which has the following signature:

```
/** * Fills an oval bounded by the specified rectangle with *
the current color. */
public void fillOval(int x, int y, int width, int height)
```

Bounding Box

- The four parameters specify a **bounding box**, which is the rectangle in which the oval is drawn. x and y specify the location of the upper-left corner of the bounding box. The bounding box itself is not drawn (see Figure B.2).
- Figure B.2: Diagram of an oval inside its bounding box.



g.setColor()

- To choose the color of a shape, invoke `setColor` on the `Graphics` object:

```
g.setColor(Color.red);
```

- The `setColor` method determines the color of everything that gets drawn afterward. `Color.red` is a constant provided by the `Color` class; to use it you have to `import java.awt.Color`. Other colors include:

```
black blue cyan darkGray gray green lightGray
magenta orange pink white yellow
```

setColor()

- You can create your own colors by specifying the red, green, and blue (**RGB**) components. For example:

```
Color purple = new Color(128, 0, 128);
```

- Each value is an integer in the range 0 (darkest) to 255 (lightest). The color `(0, 0, 0)` is black, and `(255, 255, 255)` is white.
- You can set the background color of the `Canvas` by invoking `setBackground`:

```
canvas.setBackground(Color.white);
```

Demo Program: drawing2.java

1. Set the background color of the canvas to gray color.

2. Set the graphic painter color to red. Draw a red circle at (100, 100).

3. Set the graphic painter color to blue. Draw a blue circle at (200, 200)

Demo Program: Drawing2.java

Go Blue!!!

g (Graphics object) is like a paint brush.

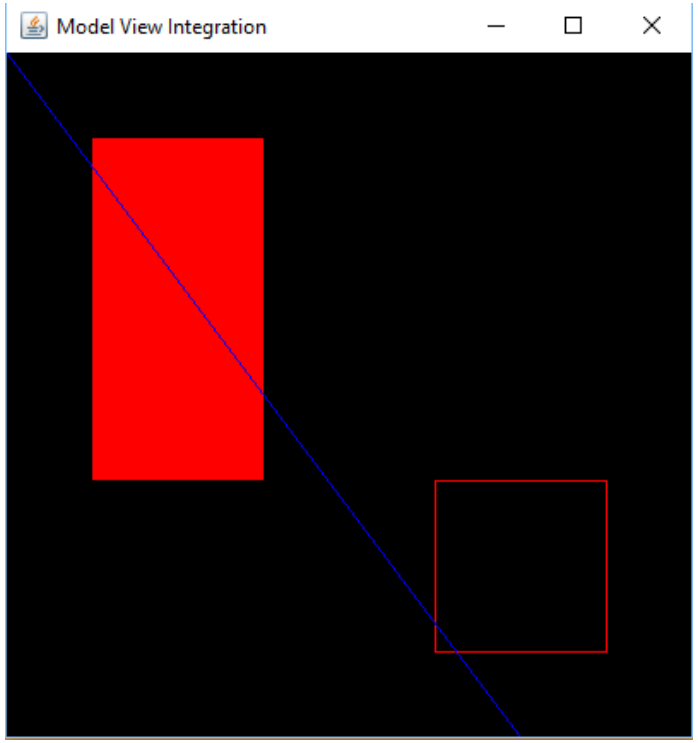
LECTURE 8

Integration of Data Model and Graphics

Integration of Data Model and Graphic View

Demo Program: MV0.java + DataModel.java

https://www.cis.upenn.edu/~bcpierce/courses/629/jdkdocs/api/java.awt.Graphics.html#_top_



Graphic View

Integration of Data Model and Graphic View

Demo Program: MV0.java + DataModel.java

Graphic View MV0.java

```
public class MV0 extends Canvas
{
    public void paint(Graphics g) {
        DataModel d = new DataModel();  // data model integrated here.
        g.setColor(Color.red);  // take red color to paint
        g.fillRect(d.box.x, d.box.y, d.box.width, d.box.height); // solid box
        g.drawRect(250, 250, 100, 100);  // empty box
        g.setColor(Color.blue);
        g.drawLine(d.p0.x, d.p0.y, d.p1.x*100, d.p1.y*100);
    }

    public static void main(String[] args){
        JFrame frame = new JFrame("Model View Integration");
        Canvas canvas = new MV0();
        canvas.setSize(400, 400);
        canvas.setBackground(Color.black);
        frame.add(canvas);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Data Model DataModel.java

```
public class DataModel
{
    public Point p1;
    public Point p0;
    public Rectangle box;

    DataModel(){
        p1 = new Point(3, 4);
        p0 = new Point(0, 0);
        box = new Rectangle(50, 50, 100, 200);
    }

    public static void main(String[] args){
        DataModel d = new DataModel();
        System.out.println(d.p1);
        System.out.println(d.p0);
        System.out.println(d.box);
    }
}
```

LECTURE 9

Mutable objects

Mutable objects

- You can change the contents of an object by making an assignment to one of its attributes. For example, to “move” a rectangle without changing its size, you can modify the x and y values:

```
Rectangle box = new Rectangle(0, 0, 100, 200);
box.x = box.x + 50;
box.y = box.y + 100;
```

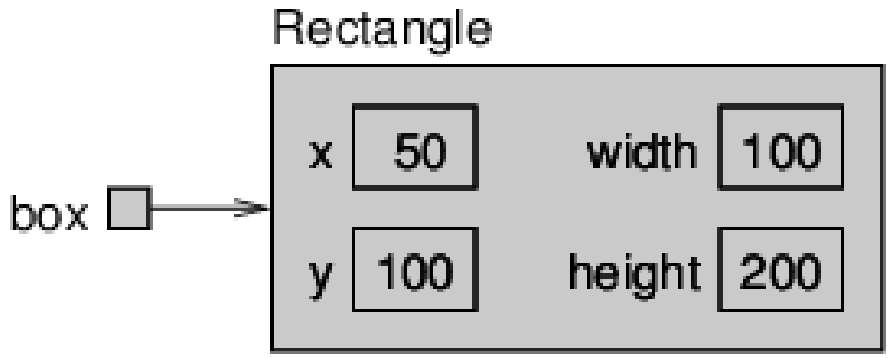


Figure 10.3: State diagram showing updated attributes.

Mutable objects

- We can encapsulate this code in a method and generalize it to move the rectangle by any amount:

```
public static void moveRect(Rectangle box, int dx, int dy) {
  box.x = box.x + dx;
  box.y = box.y + dy;
}
```

- The variables **dx** and **dy** indicate how far to move the rectangle in each direction. Invoking this method has the effect of modifying the Rectangle that is passed as an argument.

```
Rectangle box = new Rectangle(0, 0, 100, 200);
moveRect(box, 50, 100);
System.out.println(box);
```

Mutable objects

- Modifying objects by passing them as arguments to methods can be useful. But it can also make debugging more difficult, because it is not always clear which method invocations modify their arguments.
- Java provides a number of methods that operate on Points and Rectangles. For example, translate has the same effect as **moveRect**, but instead of passing the rectangle as an argument, you use dot notation:

```
box.translate(50, 100);
```

- This line invokes the translate method for the object that box refers to. As a result, the box object is updated directly.

Mutable objects

- This example is a good illustration of **object-oriented** programming.
- Rather than write methods like **moveRect** that modify one or more parameters, we apply methods to objects themselves using dot notation.

Demo Program: drawing3.java

Go BlueJ!!!

Added Features in drawing3.java

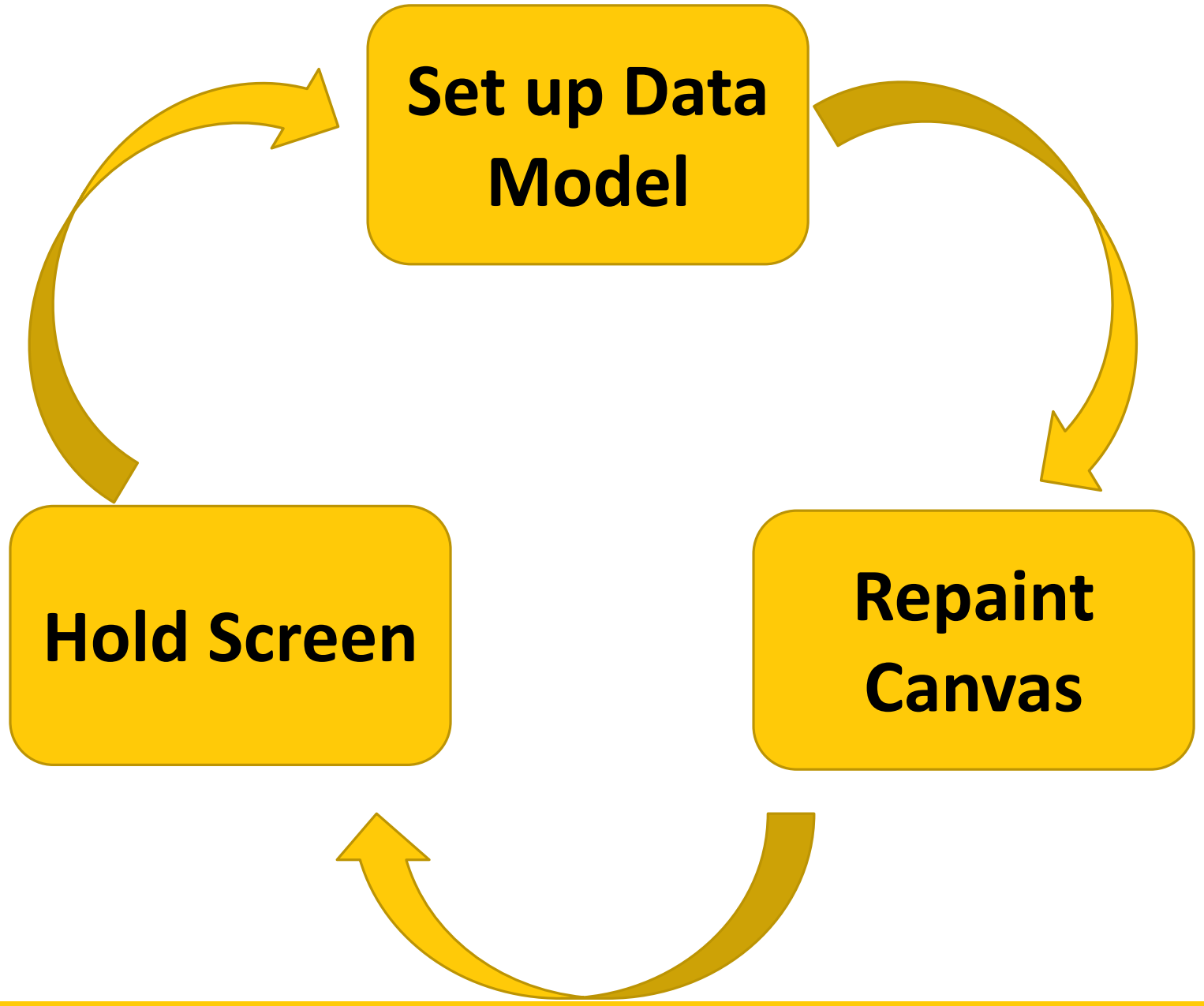
1. Use JPanel instead of Canvas (JPanel is more versatile and useful than Canvas). It is the Swing version of Canvas.
2. moveRect() method added.
3. clean screen before repaint added.
4. Infinite control loop added. (Without exit event).
5. Not-resizable window.
6. Motion (Animation added)

Simplest Animation Control Loop

Set up basic frame and canvas;
while (true){
    1. program sleep a while;
    2. update translation of objects. And all objects in the data model.
    3. repaint the canvas.
}

Note: drawing3B.java using translate method for Rectangle objects.

Loop



LECTURE 10

Aliasing

Object references (variables) are aliases

- Remember that when you assign an object to a variable, you are assigning a reference to an object. It is possible to have multiple variables that refer to the same object. The state diagram in Figure 10.4 shows the result.

```
Rectangle box1 = new Rectangle(0, 0, 100, 200);
Rectangle box2 = box1;
```

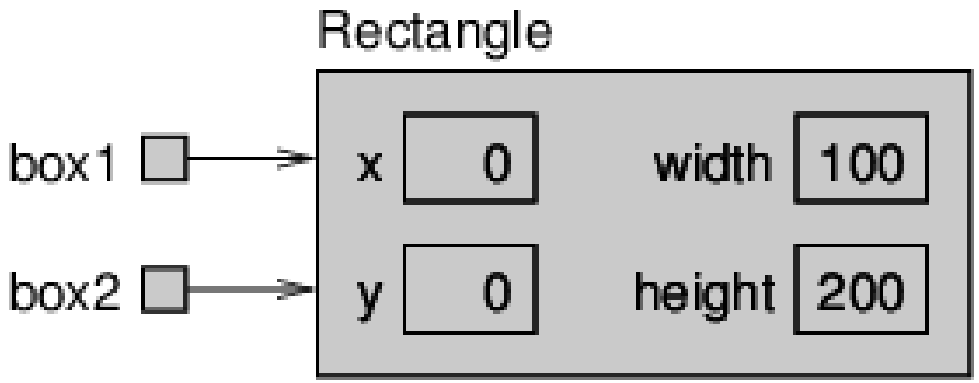


Figure 10.4: State diagram showing two variables that refer to the same object.

Aliasing

- Notice how box1 and box2 are aliases for the **same** object, so any changes that affect one variable also affect the other.
- This example adds 50 to all four sides of the rectangle, so it moves the corner up and to the left by 50, and it increases the height and width by 100:

```
System.out.println(box2.width);
box1.grow(50, 50);
System.out.println(box2.width);
```

Aliasing

- The first line displays 100, which is the width of the Rectangle referred to by box2. The second line invokes the grow method on box1, which stretches the Rectangle horizontally and vertically. The effect is shown in Figure 10.5.

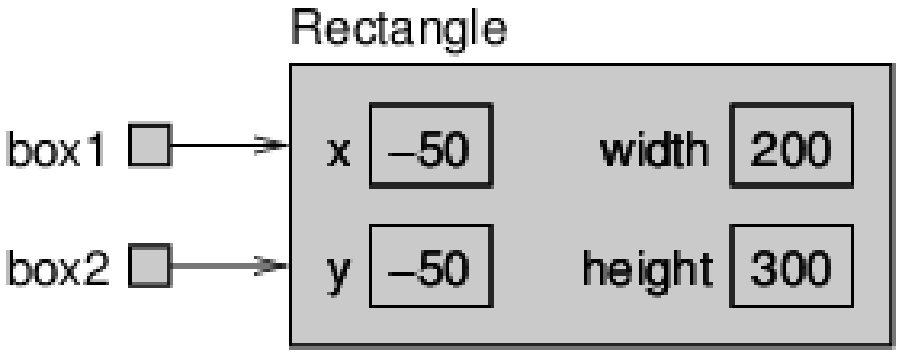


Figure 10.5: State diagram showing the effect of invoking grow.

- When we make a change using box1, we see the change using box2. Thus, the value displayed by the third line is 200, the width of the expanded rectangle

The null keyword

Initial value/Parking or Objects

- When you create an object variable, remember that you are storing a reference to an object. In Java, the keyword null is a special value that means “no object”. You can declare and initialize object variables this way:

```
Point blank = null;
```

- The value null is represented in state diagrams by a small box with no arrow, as in Figure 10.6.

Figure 10.6: State diagram showing a variable that contains a null reference.

Aliases

- If you try to use a null value, either by accessing an attribute or invoking a method, Java throws a **NullPointerException**.

```
Point blank = null; int x=blank.x; //NullPointerException
blank.translate(50, 50); // NullPointerException
```

- On the other hand, it is legal to pass a null reference as an argument or receive one as a return value. For example, null is often used to represent a special condition or indicate an error

LECTURE 11

Garbage collection

Garbage collection

- In Section 10.6, we saw what happens when more than one variable refers to the same object. What happens when no variables refer to an object?

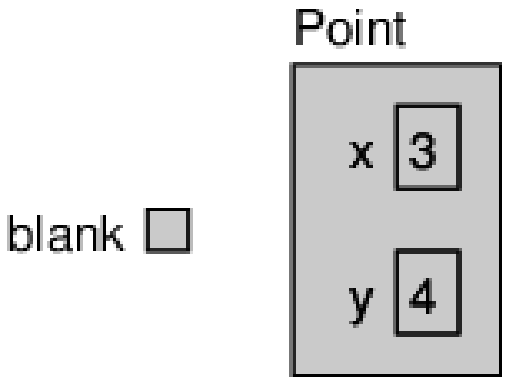


Figure 10.7: State diagram showing the effect of setting a variable to null.

- The first line creates a new **Point** object and makes blank refer to it. The second line changes blank so that instead of referring to the object, it refers to nothing. In the state diagram, we remove the arrow between them, as in Figure 10.7.

Garbage collection

- If there are no references to an object, there is no way to access its attributes or invoke a method on it. From the programmer's view, it ceases to exist. However it's still present in the computer's memory, taking up space.
- As your program runs, the system automatically looks for stranded objects and reclaims them; then the space can be reused for new objects. This process is called garbage collection.
- You don't have to do anything to make garbage collection happen, and in general don't have to be aware of it. But in high-performance applications, you may notice a slight delay every now and then when Java reclaims space from discarded objects.

LECTURE 12

Example Drawing Appendix B

Example Drawing

- Suppose we want to draw a “Hidden Mickey”, which is an icon that represents **Mickey Mouse**

  (see https://en.wikipedia.org/wiki/Hidden_Mickey).

- We can use the oval we just drew as the face, and then add two ears.

- To make the code more readable, let’s use Rectangle objects to represent bounding boxes.

Demo Program: Mickey.java

- Here's a method that takes a Rectangle and invokes fillOval:

```
public void boxOval(Graphics g, Rectangle bb) {
  g.fillOval(bb.x, bb.y, bb.width, bb.height);
}
```

Demo Program: Mickey.java

- And here's a method that draws Mickey Mouse:

```
public void mickey(Graphics g, Rectangle bb) {
  boxOval(g, bb);
  int dx = bb.width / 2;
  int dy = bb.height / 2;
  Rectangle half = new Rectangle(bb.x, bb.y, dx, dy);
  half.translate(-dx / 2, -dy / 2);
  boxOval(g, half);
  half.translate(dx * 2, 0);
  boxOval(g, half);
}
```

Demo Program: Mickey.java

- The first line draws the face. The next three lines create a smaller rectangle for the ears. We translate the rectangle up and left for the first ear, then to the right for the second ear. The result is shown in Figure B.3.

Figure B.3: A “Hidden Mickey” drawn using Java graphics.

- You can read more about Rectangle and translate in Chapter 10.
- See the exercises at the end of this appendix for more example drawings