Chapter 3: Methods in Java: Manipulating Pictures
Chapter Objectives

The computer science goals for this chapter are:

- To become more familiar with Java syntax and control statements.
- To create a variety of different kinds of methods, including those that return values.
- To recognize Javadoc comments.
- To chain method calls for compact, powerful expressions.
- To start our discussion of data structures with arrays.

The media learning goals for this chapter are:

- To extend what we can do with pictures.
- To combine methods for powerful picture manipulation.
**Assignment**

- `<Class> <variable> = <expression>;`
- `<variable> = <expression>;`
  - If the variable has already been declared.
    - You can’t declare a variable twice.
  - **Note:** In DrJava Interactions pane, variables will be declared for you.

**Style:**
- Capitalize your classnames
- Lowercase everything else
  - But can use mixed case to breakUpCombinedWords
Java: Expressions and Indentation

• In Java, statements end with “;”
  • You can use as many lines as you want, insert spaces and returns almost whenever you want. The semicolon is the end of the statement.

• Indentation doesn’t matter at all.
  • DrJava will indent for you, but just to make it easier to read.
Declaring a variable

- `<Classname> <variable>;`
- `<Classname> [] <variable>;`
- `<Classname> <variable> [];`

- With the square brackets notation, you’re declaring an array. (Turns out either way works.)
- To access part of an array, you’ll use square brackets, e.g., `myPicturesArray[5]`

I just stuck the word “Array” in there as an example. The variable could be “fred”
Expressions

• *new <Classname>(<maybe inputs>)*
  • Makes a new instance of the class
• *, /, +, -
• A shortcut:
  • \( x = x + 1 \) is so common that it can be shortened to \( x++ \)
  • \( x = x + y \) is so common that it can be shortened to \( x += y \)
Conditionals

• if (<logical-expression>)
  then-statement;
• Logical expressions are like you’d expect: <, >, 
  <=, >=, ==
  • Logical “and” is &&
  • Logical “or” is ||
• BUT then-statement can be a single statement
  **OR** any number of statements {in curly braces}. 

Conditional examples

- if (thisColor == myColor)
  setColor(thisPixel,newColor);
- if (thisColor == myColor)
  {setColor(thisPixel,newColor);};
- if (thisColor == myColor)
  {x = 12;
   setColor(thisPixel,newColor);};

You do not need these semi-colons to end the if, but they’re not wrong

Need this one to end the statement inside the curly braces
A “Block”

• We call the curly braces and the code within it a *block*.
  • A block is considered a single statement.
• A Java statement (think “sentence”) can end in a semi-colon *or* a right-curly-brace (think “.” or “!” or “?”)
Iteration: While

- `while (<logical-expression>)
  while-statement;
- You rarely will have only a single statement in a `while`, though.
- You’ll almost always have a bunch of statements in a block.
Example while

> p
Picture, filename D:/cs1316/MediaSources/Swan.jpg height 360 width 480
> Pixel [] mypixels = p.getPixels();
> int index = 0;
> while (index < mypixels.length)
  {mypixels[index].setRed(0);
   index++
  };
Error: Invalid block statement
> while (index < mypixels.length)
  {mypixels[index].setRed(0);
   index++
  };

Declaring an array of pixels
Need to have a semi-colon on the statements inside the block, too!
Side note: .length?

• Not .length()?  
  • Nope.  
  • length is an instance variable or field (different terms for same thing)  
  • It’s a variable that’s known to the instances of the class.  
  • Just as a method is a function known only to instances of the class.
Iteration: For

- `for (<initialization>; <continuing-condition>; <iterating-todo>) statement;`

- The *for* loop is unusual. It’s *very* flexible, but that means it has lots of pieces to it:
  - `<initialization>` is a statement that gets executed *once* before the loop starts.
  - `<continuing-condition>` is a logical expression (e.g., `<`, `>`, `==`) that is tested prior to each loop execution. The loop iterates only if the `<continuing-condition is true>`.
  - `<iterating-todo>` is a statement that gets executed at the end of each loop. It usually increments a variable.
Example: for

```java
for (int i=0; i < mypixels.length ; i++)
{ mypixels[i].setRed(0);}
```

• This is the same as the earlier `while` example, but shorter.
  • It sets up `i` equal to 0 to start.
  • It keeps going as long as `i` is less than the length of the pixels.
  • Each time through, it increments `i` by 1.
  • (Java oddity: `i` doesn’t exist after the loop!)
Writing Programs in Java is Making Classes

- In Java, it’s the objects that do and know things.
- So, the programming is all about defining what these objects do and know.
  - We define the variables that \textit{all} objects of that class know at the top of the class file.
  - We define the \textit{methods} for what the objects \textit{do} inside the class file.

```java
public class Picture {
    // fields
    // constructors
    // methods
}
```
Public?

• In Java, we can control what pieces of our programs other people have access to.

• Think about running a large organization.
  • You want those outside your organization accessing your company through pre-defined mechanisms: Press-releases, switchboard, technical support, salespeople.
  • You don’t want them accessing your internal intercom, internal memoranda, boardroom meetings.

• In Java, you can declare what is public and what is private (or protected for just related classes)
Visibility Controls

- *Public* visibility means that the class, field, or method is accessible by any other class.
- *Private* visibility means that the field or method can be accessed only by the declaring class.
- *Protected* visibility means access only by subclasses or classes in this package. If you ignore “packages,” ignore “protected.”
- *Package* visibility means access by any class in same package. See “protected.”
/**
 * Method to decrease the red by half in the current picture
 */

public void decreaseRed() {

    Pixel pixel = null; // the current pixel
    int redValue = 0;   // the amount of red

    // get the array of pixels for this picture object
    Pixel[] pixels = this.getPixels();

    // start the index at 0
    int index = 0;

    // loop while the index is less than the length of the pixels array
    while (index < pixels.length) {

        // get the current pixel at this index
        pixel = pixels[index];

        // get the red value at the pixel
        redValue = pixel.getRed();

        // set the red value to half what it was
        redValue = (int) (redValue * 0.5);

        // set the red for this pixel to the new value
        pixel.setRed(redValue);

        // increment the index
        index++;
    }
}
Using this method

> Picture mypicture = new Picture(FileChooser.pickAFile());
> mypicture.decreaseRed();
> mypicture.show();
> mypicture.write("D:/cs1316/ less-red-bridge.jpg");
More ways to comment

```java
/**
 * Method to decrease the red by half in the current picture
 */
```

- Anything between `/*` and `*/` is ignored by Java.
- Just like `//`, but crossing multiple lines.
A method definition

```java
public void decreaseRed()
{
    // Skipping the insides for a minute.
}
```

• Void? We have to declare the type of whatever the method returns.
  • If nothing, we say that it returns void
Variables we’ll need in this method

```java
public void decreaseRed() {
    Pixel pixel = null; // the current pixel
    int redValue = 0;   // the amount of red
}
```

- **pixel** and **redValue** are variables that are *local* to this method.
  - Give local variables values as you declare them.
- **null** literally means “nothing.”
  - If you want to put a blank value in an object variable, that’s the value to use.
- Java is *case sensitive*
  - So you can have a variable **pixel** that holds an instance of class **Pixel**.
- **int** means “integer”
More data for the method

// get the array of pixels for this picture object
Pixel[] pixels = this.getPixels();
// start the index at 0
int index = 0;

- **this? this** is how we refer to the picture (object) that is executing the method.
  - *mypicture* in the example
- **getPixels()** returns all the pixels in the object.
The loop for decreasing red

// loop while the index is less than the length
// of the pixels array
while (index < pixels.length)
{
    // get the current pixel at this index
    pixel = pixels[index];
    // get the red value at the pixel
    redValue = pixel.getRed();
    // set the red value to half what it was
    redValue = (int) (redValue * 0.5);
    // set the red for this pixel to the new value
    pixel.setRed(redValue);
    // increment the index
    index++;
}

- All arrays know their length
  - This is a reference to a variable known only to the object
- We get the pixel, then get the pixel’s red value.
- When we multiply by 0.5, we create a float
  - We say (int) to turn the value back into an integer to put in redValue.
- Then we set the pixel’s red to the new redValue.
- Finally, we move to the next pixel by incrementing the index.
/**
 * Method to decrease the red by an amount
 * @param amount the amount to change the red by
 */

public void decreaseRed(double amount)
{
    Pixel[] pixels = this.getPixels();
    Pixel p = null;
    int value = 0;

    // loop through all the pixels
    for (int i = 0; i < pixels.length; i++)
    {
        // get the current pixel
        p = pixels[i];
        // get the value
        value = p.getRed();
        // set the red value the passed amount time what it was
        p.setRed((int) (value * amount));
    }
}
What do Pictures and Pixels know?

- That’s what the JavaDoc documentation tells you.
JavaDoc

• When comments are inserted in a particular format in Java classes and methods, documentation for that class and method can be automatically generated.
• This is called JavaDoc: Java Documentation.
• It’s how Java users figure out what’s available for them to use in other classes.
  • The API: Application Programming Interface
• “What is that format?” More on JavaDoc later.
• Not all of Picture, Sound, etc. are in JavaDoc.
  • You do need to read the Picture and Sound classes, too.
Inheritance

• “But hang on a minute! The class Picture doesn’t actually know much at all!!”
• Right. *Picture inherits* from *SimplePicture*.

```java
public class Picture extends SimplePicture
```

• That means that much of what *Picture* knows and can do comes from *SimplePicture*.
• We’ll talk more about “*Why would you want to do that?*” later
Making our own methods

• Edit the .java file
• Stick your method at the bottom of the file.
  • *Inside* the final close curly brace “}” for the class.
  • Being sure to *declare* the method correctly.

• Save
• Click *Compile All*
  • Fix errors *when* they come up.
Yes, it’s scary, but change Picture.java

- If you change some other file, Pictures won’t know about your method.
- If you rename the file, it will no longer be a Picture class.
- You actually *have to* change the file we give you.
  - Don’t worry. If you screw up, you can copy down a new one.
  - Also don’t worry. The stuff that is easiest to screw up has been hidden away in SimplePicture.
Methods that return something: Compositing Images

/**
 * Method to compose (copy) this picture onto a target picture
 * at a given point.
 * @param target the picture onto which we copy this picture
 * @param targetX target X position to start at
 * @param targetY target Y position to start at
 */

public void compose(Picture target, int targetX, int targetY) {

    Pixel currPixel = null;
    Pixel newPixel = null;

// loop through the columns
for (int srcX=0, trgX = targetX; srcX < this.getWidth();
    srcX++, trgX++) {

// loop through the rows
for (int srcY=0, trgY=targetY; srcY < this.getHeight();
    srcY++, trgY++) {

    // get the current pixel
    currPixel = this.getPixel(srcX,srcY);

    /* copy the color of currPixel into target,
     * but only if it'll fit.
     */
    if (trgX < target.getWidth() && trgY < target.getHeight()) {
        newPixel = target.getPixel(trgX,trgY);
        newPixel.setColor(currPixel.getColor());
    }
}
}
Setting the Media Path

- By telling Java where you'll store your media, it gets easier to access files.

```java
> FileChooser.pickMediaPath()
The media directory is now C:\dsBook\media-source/
> String file = FileChooser.getMediaPath("mLeft1.jpg")
> System.out.println(file)
"C:\dsBook\media-source/mLeft1.jpg"
```
Trying the Compose Method

> Picture p = new Picture(FileChooser.getMediaPath("mLeft1.jpg"));
> Picture jungle =
    new Picture(FileChooser.getMediaPath("jungle.jpg"));
> p.compose(jungle,65,200);
> jungle.show();
> jungle.write(Chooser.getMediaPath("jungle-with-guy.jpg"));
How `compose()` works

• Can’t use `getPixels()`
  • We need to know where each pixel is.

• We track the source X and Y location, and copy to the target X and Y location.

```java
// loop through the columns
for (int srcX=0, trgX = targetX; srcX < this.getWidth();
    srcX++, trgX++) {

    // loop through the rows
    for (int srcY=0, trgY=targetY; srcY < this.getHeight();
        srcY++, trgY++) {

        // get the current pixel
        currPixel = this.getPixel(srcX,srcY);
```
Copy the color, if there’s room.

• Take the color from (srcX, srcY) and set it at (trgX, trgY) – as long as there’s room.

/* copy the color of currPixel into target,  
* but only if it'll fit.  
*/

if (trgX < target.getWidth() && trgY < target.getHeight()) {
  newPixel = target.getPixel(trgX, trgY);
  newPixel.setColor(currPixel.getColor());
}
Return a new picture, scaled

/**
 * Method to scale the picture by a factor, and return the result
 * @param factor to scale by (1.0 stays the same,
 * @ 0.5 decreases each side by 0.5, 2.0 doubles each side)
 * @return the scaled picture
 */

public Picture scale(double factor) {

    Pixel sourcePixel, targetPixel;
    Picture canvas = new Picture(
            (int) (factor*this.getWidth())+1,
            (int) (factor*this.getHeight())+1);

// loop through the columns
for (double sourceX = 0, targetX=0;
     sourceX < this.getWidth();
     sourceX+=(1/factor), targetX++) {
    // loop through the rows
    for (double sourceY=0, targetY=0;
         sourceY < this.getHeight();
         sourceY+=(1/factor), targetY++) {
        sourcePixel = this.getPixel((int) sourceX,(int) sourceY);
        targetPixel = canvas.getPixel((int) targetX, (int) targetY);
        targetPixel.setColor(sourcePixel.getColor());
    }
}

return canvas;
Using `scale()`

```java
> Picture blank = new Picture(600,600);
> Picture swan = new Picture(FileChooser.getMediaPath("swan.jpg");
> Picture rose = new Picture(FileChooser.getMediaPath("rose.jpg");
> rose.scale(0.5).compose(blank,10,10);
> rose.scale(0.75).compose(blank,300,300);
> swan.scale(1.25).compose(blank,0,400);
> blank.show();
```
public Picture scale(double factor)

• This scaling method returns a new instance of Picture.
  • It doesn’t change the original!
  • That will turn out to be an advantage.

• This version takes a factor for how much to scale the target picture (this)
Declaring a new picture

```java
Pixel sourcePixel, targetPixel;
Picture canvas = new Picture((int) (factor*this.getWidth()) +1,
(int) (factor*this.getHeight())+1);
```

- We need some pixels for copying things around.
- The canvas is the same size as this, but multiplied by the scaling factor, and adding one to avoid off-by-one errors.
  - The size of the Picture **must** be an int so we **cast** it into that form.
- Note: We can create new Picture instances by passing in a filename **OR** a height and width!
  - It’ll start out all-white
// loop through the columns
for (double sourceX = 0, targetX=0;
    sourceX < this.getWidth();
    sourceX+=(1/factor), targetX++)
{
    // loop through the rows
    for (double sourceY=0, targetY=0;
        sourceY < this.getHeight();
        sourceY+=(1/factor), targetY++)
    {
        sourcePixel = this.getPixel((int) sourceX,(int) sourceY);
        targetPixel = canvas.getPixel((int) targetX, (int) targetY);
        targetPixel.setColor(sourcePixel.getColor());
    }
}
return canvas;

• Anything you create in a method only exists inside that method.

• If you want it to get outside the context (or scope) of that method, you have to return it.
Manipulation without changing the original: Cascading methods

This returns a Picture —and *rose* is not changed!

```
rose.scale(0.5).compose(blank,10,10);
```

This is a method that’s understood by Pictures. Why, that’s what *scale* returns!

BTW, can use `explore()` as well as `show()` to see results!
Composing by Chromakey

/**
 * Method to do chromakey using an input color for the background
 * and a point for the upper left corner of where to copy
 * @param target the picture onto which we chromakey this picture
 * @param bgColor the color to make transparent
 * @param threshold within this distance from bgColor, make transparent
 * @param targetX target X position to start at
 * @param targetY target Y position to start at
 */

public void chromakey(Picture target, Color bgColor, int threshold,
                    int targetX, int targetY) {

    Pixel currPixel = null;
    Pixel newPixel = null;

// loop through the columns
for (int srcX=0, trgX=targetX;
    srcX<getWidth() && trgX<target.getWidth();
    srcX++, trgX++) {

// loop through the rows
for (int srcY=0, trgY=targetY;
    srcY<getHeight() && trgY<target.getHeight();
    srcY++, trgY++) {

// get the current pixel
currPixel = this.getPixel(srcX,srcY);

/* if the color at the current pixel is within threshold of
 * the input color, then don't copy the pixel
 */
if (currPixel.colorDistance(bgColor)>threshold) {
    target.getPixel(trgX,trgY).setColor(currPixel.getColor());
}
}
}
}
Chromakey replaces the background

- It’s how the weatherperson gestures to the weather map, while standing in front of a blue or green screen.

```java
> Picture p = new Picture(chooser.getMediaPath("mRight.jpg"))
> Picture back = new Picture(640,480)
> p.bluescreen(back,65,250)
> import java.awt.Color // to allows us to use the short name for color
> p.chromakey(back,Color.BLUE,100,165,250)
> p.chromakey(back,Color.BLUE,200,265,250)
> back.show()
```
blueScreen() presuming blue

/**
 * Method to do chromakey assuming a blue background
 * @param target the picture onto which we chromakey this picture
 * @param targetX target X position to start at
 * @param targetY target Y position to start at
 */

public void blueScreen(Picture target,
                      int targetX, int targetY) {

    Pixel currPixel = null;
    Pixel newPixel = null;

// loop through the columns
for (int srcX=0, trgX=targetX;
    srcX<getWidth() && trgX<target.getWidth();
    srcX++, trgX++) {

    // loop through the rows
    for (int srcY=0, trgY=targetY;
        srcY<getHeight() && trgY<target.getHeight();
        srcY++, trgY++) {

        // get the current pixel
        currPixel = this.getPixel(srcX,srcY);

        /* if the color at the current pixel mostly blue (blue value is
           * greater than red and green combined), then don't copy pixel
           */
        if (currPixel.getRed() + currPixel.getGreen() >
            currPixel.getBlue()) {
            target.getPixel(trgX,trgY).setColor(currPixel.getColor());
        }
    }
}

Note that this definition of “blue” (where red+green<blue) works better as a test than the color distance.
Some of the methods in Picture that return pictures

```java
public Picture scale(double factor)
public void chromakey(Picture target, Color bgcolor, int threshold,
  int targetx, int targety)
public void bluescreen(Picture target,
  int targetx, int targety)
public void compose(Picture target, int targetx,
  int targety)
public Picture flip()
```
How do you use all of those?

- If you were (say) to build a collage, you’d want to *use* these methods, but probably *not* in a method for Picture.
  - Individual picture objects shouldn’t necessarily be responsible for assembling lots of pictures.
- In general: How do you build a program that simply *uses* other objects?
public static void main(String [] args)

• The answer isn’t very object-oriented.
• You create a class with one method, with statements as if it were in the Interactions Pane.
  • It’s a main method, and it uses the gobbledygook above.
  • It can be run from DrJava with a menu item AND from the Command prompt
public class MyPicture {

    public static void main(String args[]) {

        Picture canvas = new Picture(600, 600);
        Picture swan =
            new Picture(FileChooser.getMediaPath("swan.jpg"));
        Picture rose =
            new Picture(FileChooser.getMediaPath("rose.jpg"));
        Picture turtle =
            new Picture(FileChooser.getMediaPath("turtle.jpg"));

        swan.scale(0.5).compose(canvas, 10, 10);
        swan.scale(0.5).compose(canvas, 350, 350);
        swan.flip().scale(0.5).compose(canvas, 10, 350);
        swan.flip().scale(0.5).compose(canvas, 350, 10);
        rose.scale(0.25).compose(canvas, 200, 200);
        turtle.scale(2.0).compose(canvas, 10, 200);
        canvas.show();
    }
}
To run it

Under Tools menu:

```
$ cd java-source
$ java MyPicture
```