

## Assignment 4—Turtle Graphics

---

In the commercial world, software systems are rarely written by a single individual. Most applications are developed by teams of programmers who work on different parts of the overall system. Each programmer is responsible for developing one or more classes, which are then combined with the other components to create the finished application.

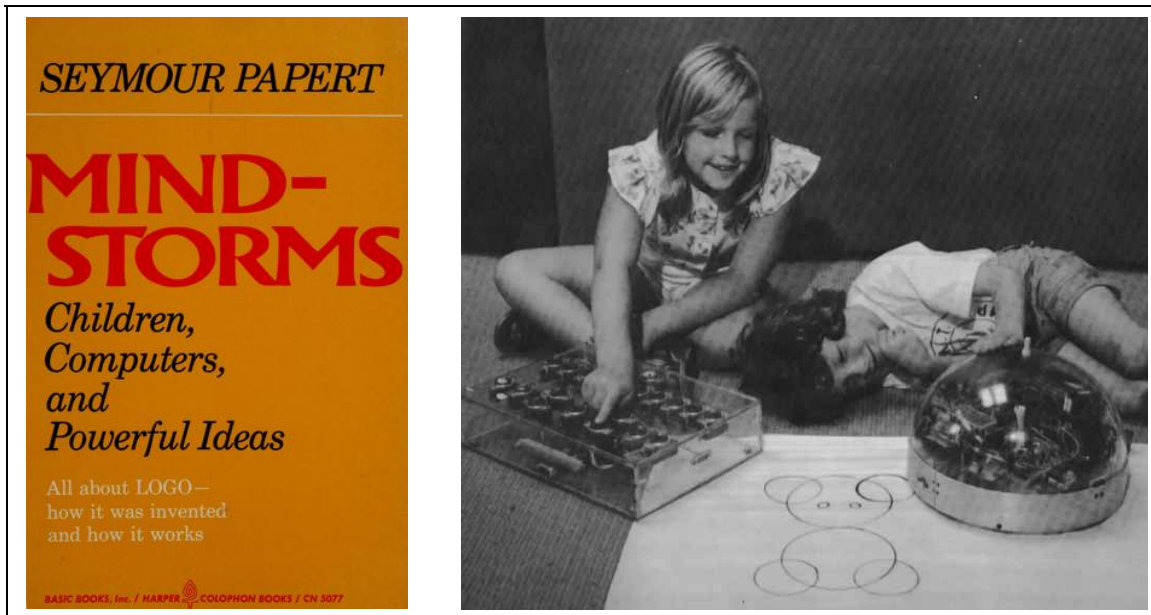
In this assignment, your job is to implement a few critical pieces of an application that lets the user program a turtle to move around the graphics window, drawing pictures as it goes. The code to display a turtle on the screen is already implemented by the `GTurtle` class, so you won't need to write that part. Moreover, because the chapters that teach you how to use interactors are still a couple of weeks in the future, we've taken care of the code to manage the user interface. The part that's left for you to implement is taking the text of the turtle program, breaking it up into individual instructions, and then translating those instructions into the appropriate method calls to the `GTurtle` object. The code you need to write involves a lot of string manipulation using the various methods provided by Java's `String` class.

### Turtle graphics

The Karel the Robot application you used at the beginning of the quarter is an example of what computer science educators call a *microworld*—a simple programming environment that is small enough for novices to master easily. Karel, however, is by no means the only microworld. Many institutions use a microworld called Alice that was developed at Carnegie Mellon by the late Randy Pausch, whose *Last Lecture* video on YouTube has been viewed more than 14 million times. Before either Karel or Alice existed, however, Seymour Papert at MIT developed the Logo language and used it to teach programming to children, who used it to maneuver a robotic “turtle” that could make drawings on paper. In 1980, Professor Papert wrote a wonderful book called *Mindstorms* that describes his experiences with the Logo turtle and offers several important insights into the dynamics of learning. Figure 1 shows the cover of *Mindstorms*, along with a picture from the book showing children enjoying the process of learning to program.

The application you will help to develop in this assignment enables users to play with a virtual turtle on the computer screen. Like the Logo turtle, the virtual turtle in this application can move forward, turn left, and turn right. In contrast to Karel's world, each of these operations takes a parameter that indicates either how far the turtle should move or how many degrees it should turn. The turtle also comes equipped with a virtual pen in its belly that it can use to draw a line on the graphics window.

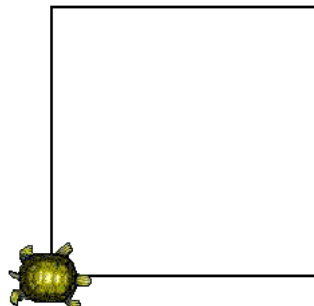
Figure 1. Cover and picture from *Mindstorms*



The programming model for the CS 106A version of the turtle is different from the Logo-based model that Seymour Papert used in his book. Partly because we need to emphasize string manipulation at this point in the course—and partly because doing so makes it easy to draw some marvelous fractal patterns—our turtle programs will be strings consisting of a sequence of commands. The individual commands consist of a single letter, which are usually followed by a number. For example, the command **F120** asks the turtle to move forward 120 pixels in the direction it is facing. The command **L90** asks the turtle to turn left 90 degrees. A program is simply a sequence of these commands. The program

```
F120 L90 F120 L90 F120 L90 F120 L90
```

moves the turtle in a square 120 pixels on a side, ending up in the same position and orientation as when it started, as follows:



The turtle language also includes the concept of repetition. If you enclose a sequence of commands in curly braces, you can repeat that entire sequence any number of times by preceding that block of commands with the letter **x** followed by the desired number of repetitions. The program to draw a square can therefore be simplified like this:

```
X4 {F120 L90}
```

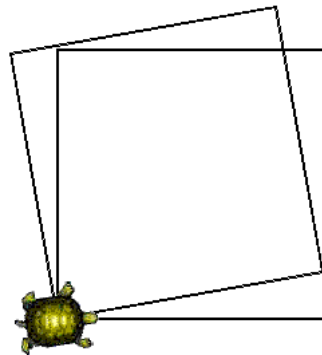
In English pseudocode form, this program therefore has the following effect:

*Repeat the following sequence of commands 4 times*  
*Move forward 120 pixels.*  
*Turn left 90 degrees.*

Repetitions can be nested to any level. You could, for example, use the program

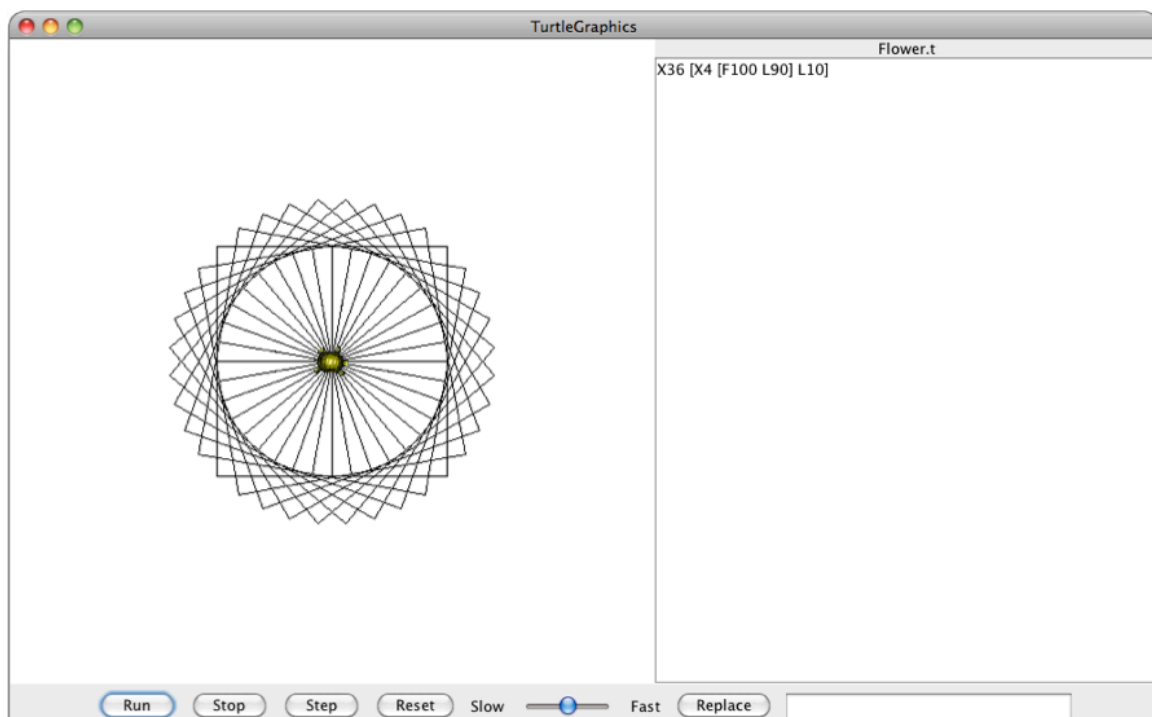
```
X2 {X4 {F120 L90} L10}
```

to draw two squares with a 10-degree left turn in the middle. The figure after drawing these two squares looks like this:



If you instead repeated the code to draw a square 36 times, the 10-degree turn will go all the way around a 360-degree circle to create the flower diagram shown in Figure 2, which comes from an example in the *Mindstorms* book.

**Figure 2. The TurtleGraphics application with a flower pattern**



## The TurtleGraphics application

The screenshot in Figure 2 shows the turtle flower program and the program that produced it in the context of the entire `TurtleGraphics` application. If you download the starter files for this assignment, load them into Eclipse, and start it up, you will see the user interface shown in Figure 2, all ready to go. You can type in turtle programs in the editor window on the right side of the window and then test those programs out by hitting the **Run** button. You can control the speed of the turtle using the speed bar and even single step the program through individual commands using the **Step** button. You can load and save files in the usual way by selecting the appropriate options from the **File** menu. In fact, the entire application works exactly as it is supposed to, including all the suggested extensions described beginning on page 14.

The fact that the application already works as soon as you download the starter files does not mean you're done with the assignment. Your job is to reimplement three parts of the `TurtleGraphics` application so that you get experience in writing classes, manipulating strings, and integrating new code into an existing application—all of which are essential programming skills. The parts you have to implement are as follows:

1. Implement the `TurtleTokenizer` class that divides a string into individual tokens.
2. Implement the `execute` method that gets the turtle to execute a program.
3. Implement the `replaceAction` method that supports a find-and-replace operation.

Each of these subtasks is described in one of the sections that follow.

### Phase 1—Implement the `TurtleTokenizer` class

Your first task in this assignment is to implement the `TurtleTokenizer` class, which is responsible for dividing up a program string into individual commands. This class is similar to the `StringTokenizer` described on page 278 of *The Art and Science of Java* and implements the same set of methods. The reason for implementing a new class is that programs in the `TurtleGraphics` application have different rules for forming tokens and therefore need a different implementation of an appropriate tokenizer class.

The most common kind of token in a turtle program is a command character (typically a letter, although any non-whitespace character is legal), which is typically followed by a sequence of decimal digits. For example, the command `F120`, which moves the turtle forward 120 pixels, is a single token in a turtle program. The digits, however, are optional. If you supply the `L` or `R` command without any digits, the `TurtleGraphics` application assumes that you want the turtle to turn 90° in the indicated direction. Similarly, executing the `F` command without any digits moves forward 50 pixels as a default. In addition, spaces are not required between tokens but are permitted for readability. These rules mean, for example, that you could rewrite the program that draws a square as

```
F120LF120LF120LF120L
```

even though doing so makes the program much more difficult for people to read. The `TurtleTokenizer` class, however, has no trouble breaking this string into substrings that

fit the definition of a token. If you use this string to initialize a `TurtleTokenizer` object, successive calls to `nextToken` should return the following sequence of eight tokens:

```
F120 L F120 L F120 L F120 L
```

For the `x` command that repeats a block of code, it is necessary to define an additional type of token. For example, the program that uses the `x` command to draw a square looks like this:

```
X4 {F120 L90}
```

When you write the code to execute this program in phase 2, you need to recognize that everything inside the braces needs to be repeated. One of the easiest ways to implement tokens enclosed in braces is to have the `TurtleTokenizer` return the entire block as a single token, so that successive calls to `nextToken` given this program should return the following token sequence:

```
X4 {F120 L90}
```

Recognizing tokens enclosed in braces is slightly harder than this example makes it seem, because it is not enough simply to collect the characters up to the next closing brace. As the `Flower.t` program in Figure 2 illustrates, the code can include nested instances of the `x` command. Thus, given the program

```
X36 {X4 {F120 L90} L10}
```

the second token should include all the characters up to the *matching* close brace, as follows:

```
X36 {X4 {F120 L90} L10}
```

The easiest strategy for finding the matching close brace is to keep a counter, adding one for each open brace and subtracting one for each close brace until the count hits zero.

Like most classes, `TurtleTokenizer` encapsulates state information along with a set of methods for manipulating that state. When you create a `TurtleTokenizer` object, the constructor has to initialize the internal state information so that subsequent calls to `nextToken` and `hasMoreTokens` can return the correct results. Thus, if you call

```
TurtleTokenizer tokenizer = new TurtleTokenizer("X4 {F120 L90}");
```

the constructor must store the string `"X4 {F120 L90}"` in an instance variable so that the object can keep track of what the input string is. In addition, the constructor needs to record the fact that the tokenizer is at the beginning of the string. If you then call

```
tokenizer.nextToken()
```

the method needs to assemble the `x` and the digits that follow it to return the token `"x4"`. In the process, the object needs to update its internal state to keep track of the fact that these characters have now been scanned. Calling `nextToken` again skips over the space, sees the opening brace, and then collect characters up to the matching brace at the end of the token `"{F120 L90}"`.

The starter code for the `TurtleTokenizer` class appears in Figure 3. Your job in this phase is to supply the implementations of the constructor and the methods `nextToken`

**Figure 3.** The starter file for the `TurtleTokenizer` class

```

/*
 * File: TurtleTokenizer.java
 * -----
 * This file implements a simple tokenizer for the TurtleGraphics system.
 */

/**
 * This class divides up a command string into individual tokens.
 * A token consists of one of two forms:
 *
 * (1) A character (usually a letter), optionally followed by any number
 *     of decimal digits, as in "F20", "R120", or "D", or
 * (2) A string beginning with "{" and continuing up to the matching "}".
 *
 * The tokenizer ignores all whitespace characters separating tokens.
 */

public class TurtleTokenizer {

    /**
     * Creates a new TurtleTokenizer that takes its input from the string str.
     * @param str The string to be scanned
     */
    public TurtleTokenizer(String str) {
        // Fill this in
    }

    /**
     * Returns true if there are more tokens to read and false if the tokenizer
     * has reached the end of the input.
     * @return A boolean value indicating whether there are any unread tokens
     */
    public boolean hasMoreTokens() {
        return false; // Replace this line with the correct implementation
    }

    /**
     * Returns the next complete token. If this method is called at the end
     * of the input, the tokenizer returns the empty string.
     * @return The next token in the input
     */
    public String nextToken() {
        return ""; // Replace this line with the correct implementation
    }

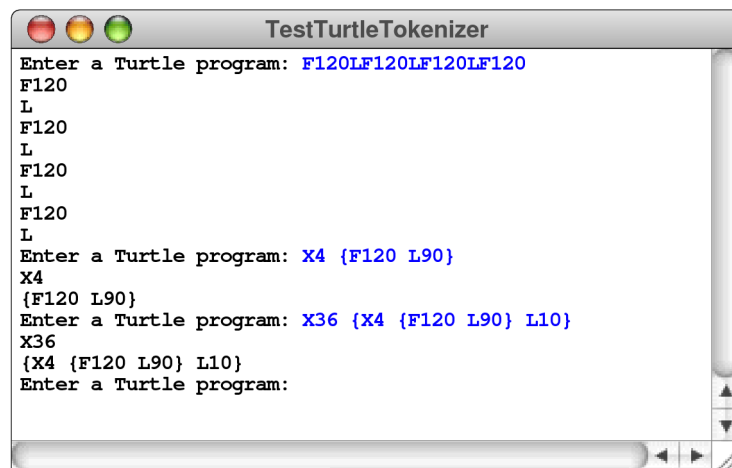
    // Add private methods and instance variables here
}

```

and `hasMoreTokens` so that the `TurtleTokenizer` class behaves as described in the earlier parts of this section. To do so, you will have to declare a few instance variables, and you may find it helpful to define a private method or two. You should not, however, have to write a lot of code. The whole of the implementation will almost certainly fit on two pages.

Even though the implementation of `TurtleTokenizer` is short, you still have to test and debug it. One approach—not the one I recommend—is to charge ahead to Phase 2 and debug both phases together. That, however, is not how professional programmers work. Once you have coded the `TurtleTokenizer` class, you really need to test it while the code is fresh in your mind. What’s more, if you don’t already have code that tests the class, you have to write a small program that makes sure that everything in your class is working correctly.

As part of this assignment, you are required to create a new class (just use the **New** function under the **File** menu and select the **Class** option) called `TestTurtleTokenizer` that reads in lines of text and then uses the `TurtleTokenizer` class to print out the tokens one per line. The output of your program should look like this:



You should use the `TestTurtleTokenizer` to test as many examples as you can to make sure all the special cases are working. This sample run tests commands with no separating spaces along with those that have them, commands without arguments like `L` and commands that do like `F120` or `x4`, and examples with nested braces.

### Phase 2—Implement the `execute` method

Once you are satisfied that you have the `TurtleTokenizer` class working, you need to turn your attention to the `TurtleGraphics` application itself. The starter file for the application appears in Figure 4. The constructor begins by creating a `GTurtle` object. It then creates an instance of a class called `TurtleGraphicsUI`, which is responsible for all the user-interface activity: the buttons, the speed bar, the menus, and the program window. You’ll have a chance to build your own user interface in Assignments #5 and #6, but that work is done for you here. The only things you have to do are write the bodies of the two missing methods, starting with the implementation of `execute`.

Figure 4. The starter file for the TurtleGraphics application

```

/*
 * File: TurtleGraphics.java
 * -----
 * This file represents the starter file for the TurtleGraphics application.
 * Your job in this assignment is to complete the application by reimplementin
 * the execute and replaceAction methods so that they no longer calls the stub
 * methods in the TurtleGraphicsUI class. You also need to implement the
 * TurtleTokenizer class in the file TurtleTokenizer.java.
 */

public class TurtleGraphics {

    /*
     * Runs the application by creating a new TurtleGraphics instance.
     */
    public static void main(String[] args) {
        new TurtleGraphics();
    }

    /*
     * Initializes the TurtleGraphics application.
     */
    public TurtleGraphics() {
        turtle = new GTurtle();
        ui = new TurtleGraphicsUI(this, turtle);
        ui.start();
    }

    /*
     * Executes a turtle program, which consists of a string of commands.
     * Each command consists of a single letter, optionally followed by an
     * integer. The commands you must implement for this assignment are
     *
     * F#      Moves forward the specified number of pixels (default = 50)
     * L#      Turn left the specified number of degrees (default = 90)
     * R#      Turn right the specified number of degrees (default = 90)
     * U       Raise the pen so that moving no longer draws a line
     * D       Lower the pen to resume line drawing
     * X#{cmds} Execute the block of commands the specified number of times
     */
    public void execute(String str) {
        ui.execute(str);        // Replace this call with your own code
    }

    /*
     * Implements the Replace button action, which replaces all instances of a
     * pattern string in the current file with a replacement string. The
     * pattern and replacement strings are taken from the replacement field in
     * the user interface, where they appear as pattern->replacement.
     */
    public void replaceAction() {
        ui.replaceAction();    // Replace this call with your own code
    }

    /* Private instance variables */

    private GTurtle turtle;        /* The GTurtle object        */
    private TurtleGraphicsUI ui;    /* The user-interface object */
}

```



**Figure 5. Useful methods in the GTurtle class**

<code>GTurtle turtle = new GTurtle();</code> The constructor creates a new <code>GTurtle</code> object facing east.
<code>turtle.penDown()</code> Lowers the pen so that moving causes the turtle to draw a line.
<code>turtle.penUp()</code> Raises the pen so that moving no longer draws a line.
<code>turtle.forward(n)</code> Moves the turtle <i>n</i> pixels forward in the direction it is facing, drawing a line if the pen is down.
<code>turtle.left(angle)</code> Turns the turtle <i>angle</i> degrees to the left.
<code>turtle.right(angle)</code> Turns the turtle <i>angle</i> degrees to the right.

The `GTurtle` class is one of the subclasses of `GObject`, and is therefore similar in some respects to the more familiar graphics classes like `GRect`, `G Oval`, and `GLine`. The `GTurtle` class, however, extends the standard behavior of the `GObject` class by adding several methods designed to support the `TurtleGraphics` style of drawing used in this assignment. You can find descriptions of the complete set of methods available for the `GTurtle` class in the **javadoc** documentation for the ACM Java libraries. To make things easier, I've included short descriptions of the methods you need for this assignment in Figure 5.

The `execute` method has the responsibility of taking a string representing a turtle program and translating that string into the appropriate method calls for the `GTurtle` object. For example, given the program string

`"F120 L90 F120 L90 F120 L90 F120 L90"`

the `execute` method will have to

1. Divide the string into tokens using the `TurtleTokenizer` class.
2. Translate each token into the appropriate method call to the `GTurtle` object. Thus, executing the `F120` token needs to invoke the method call `turtle.forward(120)`. Similarly, executing the `L90` token needs to invoke a call to `turtle.left(90)`.

Your program is required to implement the command forms shown in Figure 6.

**Figure 6. Required commands for the TurtleGraphics application**

<code>F<i>n</i></code>	Move the turtle forward <i>n</i> pixels. If <i>n</i> is missing, you should assume a <i>default value</i> of 50.
<code>L<i>n</i></code>	Turns the turtle left <i>n</i> degrees, where <i>n</i> defaults to 90.
<code>R<i>n</i></code>	Turns the turtle right <i>n</i> degrees, where <i>n</i> defaults to 90.
<code>D</code>	Calls <code>penDown</code> so that the turtle draws a line. In <code>TurtleGraphics</code> , the pen starts down.
<code>U</code>	Calls <code>penUp</code> so that the turtle stops drawing a line.
<code>X<i>n</i> {cmds}</code>	Repeats the specified block of commands <i>n</i> times, where <i>n</i> has a default value of 2.

The first five commands in Figure 6 have straightforward implementations. For the **F**, **L**, and **R** commands, you have to check to see if the command letter is followed by an integer. If so, you have to call `Integer.parseInt` on the string of digits to convert it into an `int` value. If not, you have to choose the correct default value for the command and use that value instead. The task of checking to see whether a token has an explicit value and supplying a default value if it is missing occurs often enough that it makes sense to define a helper method to accomplish this task. Once you have the argument value, all you have to do is call the appropriate method on the `GTurtle` object, which is stored in the instance variable `turtle`.

At first glance, it seems as if implementing the **x** command would be substantially harder, but the difficulty goes away if you think about the problem creatively. Obtaining the argument *n* uses the same helper method that you wrote for the **F**, **L**, and **R** commands. Reading the block is also easy because the `TurtleTokenizer` class returns a string enclosed in curly braces as a single token. Repeating an operation *n* times requires nothing more than the classic `for` loop pattern. The only question is how to execute the string of commands enclosed inside the curly braces. That task would be hard if you started all over again and rewrote the code to execute a command string. The good news is that you already have a method that executes a command string. It's called `execute`. To implement this part of the **x** command, all you have to do is take the curly braces away from each end of the token and call `execute` on the string that sits between them.

Including a call to `execute` inside the definition of `execute` is an example of a technique called *recursion*, which is the process of solving a problem by transforming it into a simpler instance of the same problem type. I touched on recursion in the Karel booklet, and you will see lots more of it if you continue on to CS106B. I chose to include it in this assignment because this particular application of recursion seems so natural and straightforward. If you ignore for the moment how Java might implement this technique and just write the call to `execute` in your implementation of the **x** command, everything will just work. And when it does work, you may take away the idea that this scary notion of recursion is not so hard after all.

As you implement the **x** command, there are a few additional requirements that you should keep in mind:

- Your implementation should recognize command characters without regard to case. Thus, the commands `F120` and `f120` should both move the turtle forward 120 pixels.
- Your code should ignore any command characters that it doesn't recognize. Arguably, it would be better to generate an error message when this occurs, but reporting those errors to the user gets a bit tricky. Moreover, being able to include letters that don't stand for any operation but can nonetheless appear in search-and-replace operations turns out to be useful in creating complex `TurtleGraphics` patterns.

### Phase 3—Implement the `replaceAction` method

As you can see in Figure 2, the `TurtleGraphicsUI` class creates a *control strip* at the bottom of the window that contains buttons to control the execution of the program and a slider to adjust the speed. It also contains a **Replace** button that triggers a call to the

`replaceAction` method in the `TurtleGraphics` application. The effect of that button is to replace every instance of one string appearing in the program by a different string. The first string is called the *pattern*, and the second is called the *replacement*. Both the pattern and the replacement are entered in the text field to the right of the **Replace** button in the form

*pattern -> replacement*

where the `->` symbol used to indicate an arrow is composed of the characters `-` and `>` written together. Whitespace characters should be ignored around the arrow or at the beginning or end of the *pattern* and *replacement* strings, but not in their interiors.

As a simple example of how you might use the **Replace** button, suppose that you want to double the size of the square drawn by the original version of the `Square.t` program, which looks like this:

```
F120 L90 F120 L90 F120 L90 F120 L90
```

If you enter the string

```
F120 -> F240
```

in the text box and then click the **Replace** button, the application should change the program to

```
F240 L90 F240 L90 F240 L90 F240 L90
```

which would indeed double the size.

To implement the `replaceAction` method, your program needs to communicate with the user interface using the methods shown in Figure 7. The code for `replaceAction` must call `getReplacementField` to get the contents of the text field, use string operations to find the `->` symbol and separate out the *pattern* and *replacement* fields, call `getProgramText` to get the current text of the program, perform the replacement operations, and finally call `setProgramText` to update the program.

**Figure 7. Public methods in the `TurtleGraphicsUI` class**

<pre>ui = new TurtleGraphicsUI(this, turtle);</pre>
The constructor creates a new user interface for the <code>TurtleGraphics</code> application. The arguments are the application and the <code>GTurtle</code> object created by the starter code.
<pre>ui.start();</pre>
Starts the user interface. This method is called by the starter code.
<pre>String text = ui.getProgramText();</pre>
Returns the entire text of the current program as a string.
<pre>ui.setProgramText(str);</pre>
Replaces the current program with the new string.
<pre>String replacement = ui.getReplacementField();</pre>
Returns the contents of the replacement field.

The `replaceAction` method should replace every instance of the *pattern* string with the *replacement* string in exactly the same way that replace-all operations work in a word processor or program editor. The match may occur at any position in the string and pays no attention to token boundaries. Thus, if you apply the replacement pattern

```
F1 -> F2
```

to the program

```
F120 L90 F120 L90 F120 L90 F120 L90
```

your program should produce the program

```
F220 L90 F220 L90 F220 L90 F220 L90
```

If you explore the documentation for Java’s `String` class, you’ll discover that the class contains a `replaceAll` method that initially *seems* to do exactly what you want. Unfortunately (at least for this assignment), the first argument to the built-in version of `replaceAll` is not actually a string but something called a “regular expression.” In regular expressions, the brace characters used for repetition (and the bracket characters described in the list of extensions) are treated specially, which means that replacement operations involving these characters would fail. What you therefore need to do is implement your own version of the “replace-all” operation.

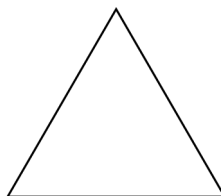
### Drawing fractals

The simple example from the preceding section of doubling the size of a square only scratches the surface of what you can do with the **Replace** button in the `TurtleGraphics` application. One of the most important advantages of representing turtle programs as strings is that you can use repeated substitution to draw *fractals*, which are shapes that incorporate the same basic structure at different scales. The word *fractal* was introduced by the late Benoit Mandelbrot in his 1975 book *The Fractal Geometry of Nature*, which generated considerable interest across a variety of fields.

To get a sense of the relationship between fractals and turtle graphics, consider the program `Triangle.t`, which draws an equilateral triangle whose sides are 81 pixels long:

```
X3 {F81 L120}
```

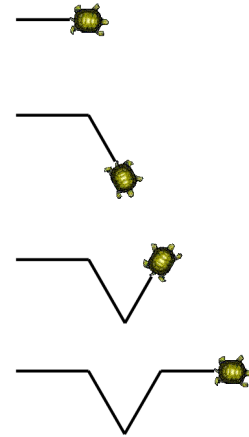
This program causes the turtle to draw the following triangle on the screen:



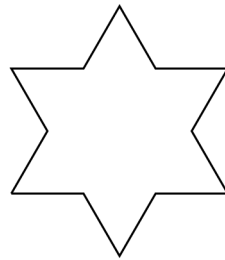
The first step in turning this diagram into a fractal is to replace each of the lines in this figure with one that contains an angular wedge. The turtle draws a single edge in the original figure by moving forward 81 units. To create the first level in the fractal design,

the turtle instead connects the same endpoints by executing the following four steps, leading to the configuration shown on the right after each step:

1. Move forward one-third of the total distance, which in this case means moving forward 27 pixels (81 divided by 3).
2. Turn  $60^\circ$  to the right and move forward 27 pixels.
3. Turn  $120^\circ$  back to the left and again move forward 27 pixels.
4. Turn  $60^\circ$  to the right and advance 27 pixels to reach the end.



If you perform this same transformation on all three sides of the triangle, you get the following diagram, which is called a **Koch snowflake** after its inventor, the Swedish mathematician Helge von Koch:



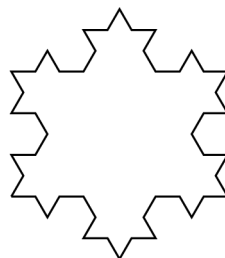
To create this diagram, all you have to do is make the following global replacement:

**F81 -> F27 R60 F27 L120 F27 R60 F27**

The figure is called a Koch snowflake of **order 1** because it has undergone only one such transformation. If you then activate the **Replace** button with the replacement string

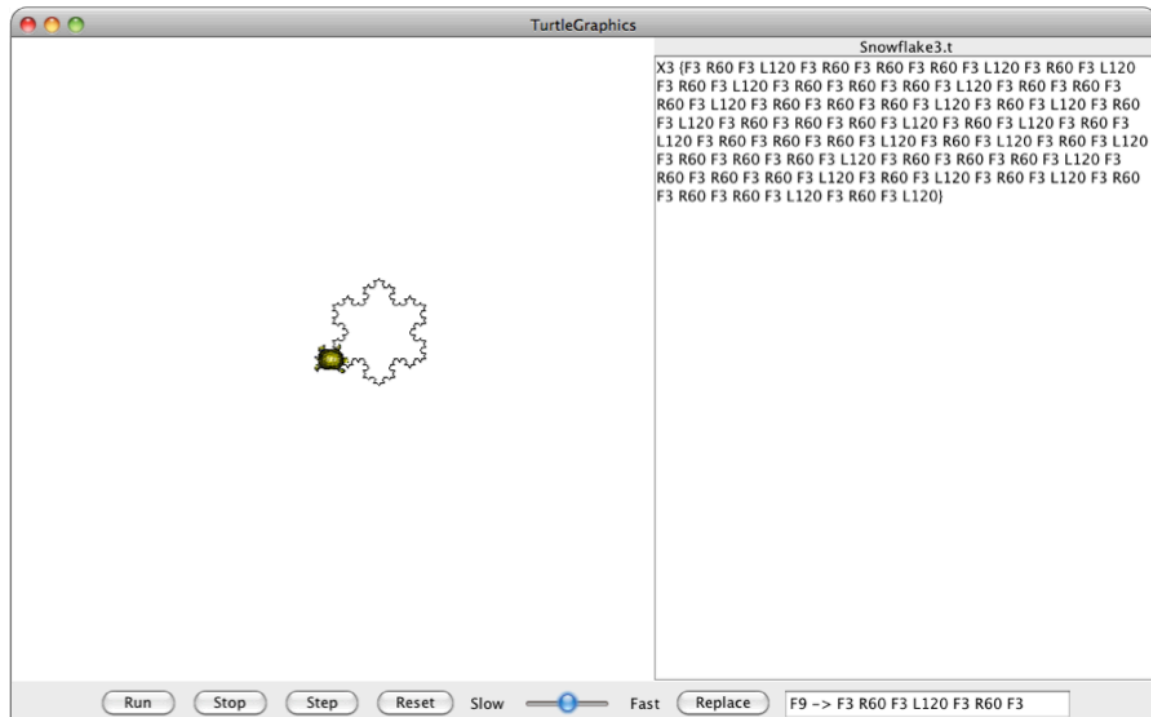
**F27 -> F9 R60 F9 L120 F9 R60 F9**

you get the Koch snowflake of order 2:



Repeating this transformation yet again creates the order-3 snowflake shown in Figure 8.

Figure 8. Screen shot showing the Koch snowflake of order 3

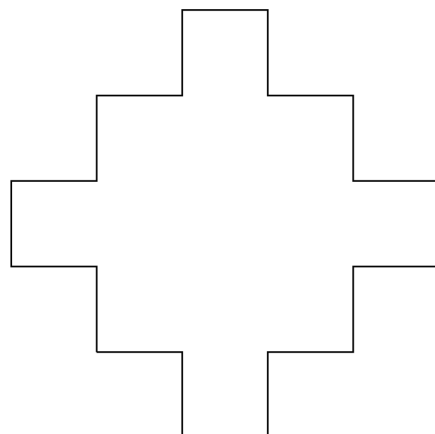


As you can see from Figure 8, the turtle program has become very long and complicated. The good news is that nobody had to type that program in; all that the user needed to do was enter the code for the original equilateral triangle and then apply a series of simple transformations.

You can make similar fractal designs starting with other figures as well. For example, if you apply the transformation

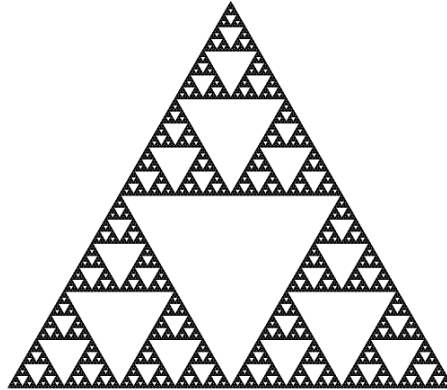
**F120 -> F40 R90 F40 L90 F40 L90 F40 R90 F40**

to the program that draws a 120-pixel square, you get the following diagram:



You can use the **TurtleGraphics** application to create most of the best-known fractal designs. The **programs** folder in the starter project, for example, includes the code to

draw the fractals you have seen earlier in this section, as well as one to draw the following figure, which is called a *Sierpinski triangle*:



Be sure to play with some of these figures as you write your assignment. Not only will it increase your fun, but you'll also have a chance to win a special grand prize—that perfect score on the final or any other single grade component—as described in Handout #34.

### Error handling

One issue that has been ignored so far in this handout is the question of how your program should handle errors. A significant part of most applications consists of handling error conditions correctly. In the `TurtleGraphics` application, your code should test for the following error conditions:

- If the `TurtleTokenizer` class tries to read a token that begins with a curly brace but cannot find a matching close brace character, the tokenizer should report the error `"Missing closing brace"`.
- If the `x` command is not followed by a block token, your implementation of `execute` should report the error `"Missing command block"`.
- If the contents of the replacement field does not contain the substring `"->"`, your implementation of `replaceAction` should report the error `"Missing ->"`.

In each of these cases, your program should report the error by including a line like

```
throw new RuntimeException(message);
```

where *message* is the error message you want to display. The `throw` statement is part of Java's facilities for *exception handling*, which you will learn more about later in the quarter.

It is useful to note at this point that your program should not report undefined command letters as errors but should simply ignore such commands if they appear.

### Extensions

There are many extensions that you could add to the program. Most of these take the form of new commands. The starter file implements the extended set shown in Figure 9.

**Figure 9. Extended command set supported by the starter program**

<b>M</b> <i>n</i>	Changes the default move step (initially 50) to the integer <i>n</i> .
<b>A</b> <i>n</i>	Changes the default angle used in the <b>L</b> and <b>R</b> commands (initially 90) to the integer <i>n</i> .
<b>H</b>	Hides the turtle by calling <code>turtle.hideTurtle</code> .
<b>S</b>	Makes the turtle visible by calling <code>turtle.showTurtle</code> .
<b>Cr:b:g</b>	Changes the color of the pen used to draw the path. The parameters <i>r</i> , <i>b</i> , and <i>g</i> are integers in the range 0 to 255 indicating the amount of red, blue, and green in the color, respectively. You can change the color of the pen in the <code>GTurtle</code> class by invoking <code>turtle.setColor</code> with a Java color.
<b>P</b> <i>n</i>	Pauses execution for <i>n</i> milliseconds. You can implement this command by calling <code>pause(n)</code> , which is defined in the <code>GraphicsProgram</code> class.
[	Saves the current state of the turtle, which consists of its position, orientation, pen state (up/down), pen color, and visibility (show/hide) so that those values can be restored by a subsequent ] command.
]	Restores the state of the turtle saved by the most recent [ command. The [ and ] can be nested to any depth, so you will need to come up with some way of saving a list of all previous saved states and restoring the most recent one.

Another useful extension is to change the implementation of `replaceAction` so that it supports multiple substitutions. In the starter program, this feature is implemented by allowing the replacement field to contain several individual substitution rules separated by commas. Thus, the substitution

**L -> R, R -> L**

reverses the direction of every turn in the program, effectively creating a mirror image of the original figure. It is important to note that these substitutions must be executed *simultaneously* rather than sequentially. If you first replaced all occurrences of **L** with **R** and then all occurrences of **R** with **L**, you would end up having only left turns in the program.

Adding this extension to `replaceAction` allows the `TurtleGraphics` application to implement *Lindenmayer systems*, which are defined by this ability to perform multiple simultaneous substitutions. Named for the Hungarian biologist Aristid Lindenmayer (1925–1989), these systems make it possible to generate fractal designs that model the growth of plants, along with the classical fractal forms from mathematics. If you look up the Wikipedia article on Lindenmayer systems, you will see many exciting examples of these techniques.