Introduction to Interactive Programming
by Lynn Andrea Stein
A Rethinking CS101 Project

Interlude: A Community of Interacting Entities

Overview

This interlude provides a whirlwind introduction to most of the basic concepts of Java programming. It uses a simple community of word games and other String transformers to illustrate this exploration.

This interlude is not intended to be read as standalone coverage of these ideas. Instead, it introduces many concepts only briefly, but in context. Each of the programming concepts presented here is reintroduced in much greater detail in the chapters of section 2 of this book.

Objectives of this Interlude

1. To increase familiarity with the design process.
2. To understand how to describe a system design in terms of types, components, and interactions.
3. To discover how design translates into executable code.
4. To be able to read and begin to understand fragments of Java programs.

Introduction: Word Games

When I was a child, we used to amuse ourselves by speaking to one another in a special language called Pig Latin. The simplest version of Pig Latin has just one rule: To turn an English word into a Pig Latin one, you take the first letter off the word, then add the first letter plus "ay" to the end of the word. So, for example, "Hello" in Pig Latin is "ello-Hay", and "How have you been?" is "ow-Hay ave-hay ou-yay een-bay?" There are more sophisticated rules for Pig Latin that deal with consonant blends and words that begin with vowels, but the basic idea remains the same. It turns out that there are children's games like Pig Latin in many, many languages, though each has a slightly different set of rules. Another such game, popularized by the children's Public Television show Zoom, is Ubby Dubby, in which you add "ubb" before every vowel (cluster): "Hubbellubbo", "Hubbow hubbave yubbou bubbeen?"

This interlude explores such word- and phrase- transformations. In fact, we're going to build a system in which you can have many of these different transformers, and you can glue them together in almost any order. In this sense, the transformers will be interconnectable modules like Lego(tm) or Capsela(tm).

[Insert pictures of various objects here.]

In addition to transformers such as Pig Latin and Ubby Dubby, we'll want capitalizers ("HELLO"), name droppers ("Lynn says Hello", or "Chris says Hello", or "Pat says How are you doing?"), even delayers (e.g., that don't produce "Hello" until after they've already received "How are you doing?") or network-

senders (that can move one of these strings-of-words from one computer to another). We'll also have some community members that can read information that a user types to them or display information on a computer screen. And we'll have transformers that can take listen to two different inputs, producing only one output, as well as transformers that can produce two outputs from only one input. (The first of these is a collector; the second is a repeater. The first is good when you have lots of people trying to talk all at once; the second is a nice way to circulate (or broadcast) information that needs to get to a lot of people.)

In the system that we're going to explore, we will need a way to create individual transformer-boxes like the ones described above. We'll also need a way to connect them together. Finally, the transformer-boxes will need to act by themselves, to read inputs, do transformations, and produce outputs. The complete system will be a community of interacting entities, many of which will themselves be communities. At the most basic level, each of these entities will need to follow specific instructions. In this interlude, we will explore both the design of the community and the specific instructions that some of these entities will follow.

Designing a Community

We need to design

- What behavior does the system provide?
- Who are the members of the community?
- How do they interact?
- What goes inside each one?

We can start at the bottom (**bottom-up design**) or at the top (**top-down design**). Both are legitimate and useful design techniques. However, design in practice often mixes these techniques. In this case, we're actually going to start in the middle; in this particular system, that is one of the easiest places to begin thinking about what we want to produce.

At the end of the design process, we should be able to sketch out a scenario for each of the major interactions with our system, including what roles need to be filled (i.e., the types of things in our system), who fills these roles (i.e., the individual objects that make up the system), and how they communicate among themselves (i.e., the flow of control among these objects).

A Uniform Community of Transformers

There are several communities implicit in the system that we're building. Let's start in the middle, where the system can be understood as a community of interacting transformers. In this picture, each transformer is an entity. The interactions in this community are quite simple: Each string transformer reads in a phrase and writes out a transformed version of it. In this system, we want to be able to interconnect these transformers in arbitrary ways. This means that the services each transformer provides will need to be compatible, so that one transformer can interact with any other transformer using the same connection mechanism.

Transformer Entity interactions, version 1

- Read a word/phrase (from a connection)

- Write a word/phrase (to a connection)

We will accomplish this generic connection between transformer entities using a computer analog of the tin can telephones that we built as children.[Footnote: Take two tin cans with one end removed from each. Punch a whole in the center of the intact end of each can. With a long piece of string, thread the two cans so that their flat ends face each other. Tie knots in the ends of the string. Pull the string tight, so that it is stretched between the two cans. Talk into one can; have someone else listen at the other.] This is a simple device that allows you to put something in one end and allows someone else to retrieve it at the other end. The computer analog will be Connection objects that allow one transformer to write a word or phrase and another transformer to read it from the connection. The transformers on either end don't have to know anything about one another; they can simply assume that the transformers will interact appropriately with the Connection. And the connections don't have to know much of anything about the transformers, either

Connection Entity interactions

- Accept a word/phrase written to you
- Supply a word/phrase when requested (read)

Connections provide one particular way of providing interconnections among objects. In this system, the components are designed so that any outputter can be connected to any inputter. In other parts of this book, we will see examples of other kinds of interaction mechanisms. For example, in some systems, the pieces to be interconnected are not uniform. In others, the particular choices of interconnections must be made at the time that the system is designed rather than while the system is running. In part 4 of this book, we will pay particular attention to the tradeoffs implicit in different interconnection mechanisms.

The User and the System

Before we look at how each transformer (and connector) is built, let's step back from this community of interacting transformers to ask how it came into existence. At this level, the members of our community are the user who constructs the community and the system to be constructed. The user expects the system to provide a way to create transformer entities and a way to connect them.

System/User interactions

- Create a Transformer (of a specified type)
- Connect two Transformers (in a particular order)

[Picture of Control Panel & transformers.]

We'll accomplish the first of these by adding another entity to the community: a user interface containing a control panel that allows the user to specify that a transformer should be created as well as what type of transformer it should be. The second interaction, connecting transformers, we will handle by letting the user specify two transformers (through the user interface) and then asking the specified transformers to accept a new connection. So allowing the system to interact with the user creates one additional entity (the user interface) and adds an interaction to the transformer:

User Interface interactions

- Create a Transformer (of a specified type)
- Create a Connection between two Transformers

Transformer Entity interactions, version 2

- Accept an input Connection [Footnote: Maybe more than one.]
- Accept an output Connection
- Read a word/phrase (from a connection)
- Write a word/phrase (to a connection)

[Picture of user interaction flow: click button -> create transformer,

click transformers -> create connection & request transformers accept it .]

Specifically, the Control Panel will have buttons representing each kind of transformer available. Clicking on a button will create a new transformer of the appropriate type. Clicking on first one transformer, then another, will create a connection between them. This task is actually cooperative: the user interface will create the connection and it will ask the Transformers to accept it.

What Goes Inside

[Insert control flow pic for transformer creation, control flow pic for string transformation interaction.]

In the two subsections immediately above, we've designed transformer-transformer interactions (via connections) and user-system interactions (via the user interface). We've addressed the question of who our community members are (UI, transformers, connections, and -- stepping back -- the user) and, to a first approximation, how they interact. In terms of system design, transformers and connectors represent kinds of things of which there may be many separate instances. For example, a particular community of transformers may contain five transformers and four connectors, or eight transformers and three connectors, or twelve. Each community will contain only a single control panel, though.

The next step in a full design process would be to look inside each of these entities to discover whether they are, themselves, monolithic or further decomposable into smaller communities. We will not decompose the user interface further in this chapter; much of the necessary background for this task will not be introduced until part 3 of this book. Instead, the remainder of this interlude will look inside the transformer type to see how these objects are built.

Building a Transformer

We have seen above the specification of the interactions that a Transformer Entity will be expected to fulfill. We can turn this interaction specification around to provide a specification of the behavior that an implementation will need to satisfy: A Transformer must be able to:

- Accept an input Connection
- Accept an output Connection
- Have its own instruction-follower that acts independently to read its input, transform that input as appropriate, and write its output.

In fact, this Transformer is itself a community. The connection acceptors are each entities that are activated only on a connection accept request; their jobs are to remember the connections that they have been handed. For example, the acceptInputConnection instructions basically say, "To accept an input connection (let's call it `in`), simply store `in` away somewhere so that you can use it later." There's also a little bit of additional code to say what to do if you've already got an input connection stored away. Output connections -- another part of the community inside an individual Transformer -- are handled in the same way as input connections. Also, some kinds of Transformers will have code that needs to be run when an individual Transformer is created. Finally, the independent instruction-follower is an additional ongoing interacting entity. It makes use of the connections (such as `in`) that the connection-acceptors have stored. Each transformer will have its own instruction follower, allowing the transformer to do its work without any other entity's needing to tell it what to do.

For the moment, we will focus on the heart of the Transformer, the work done by this independent instruction-follower, especially the transformation it actually performs. We begin by looking at some specific Transformers and describing the behavior we expect.

Transformer Examples

The instructions for the behavior of a Capitalizer will say

1. Read the input.
2. Produce a capitalized version of it.
3. Write this as output.

Every individual Capitalizer is the same, and each one does the same thing. You can tell them apart because they're connected to different parts of the community and are capitalizing different words, though.

NameDropper is a different kind of Transformer. Each individual NameDropper has its own name that it likes to drop. So the instructions for a NameDropper will say

1. Read the input.
2. Produce a new phrase containing your name, the word "says", and the input.
3. Write this as output..

Variations in Transformer behavior aren't restricted to the transformation itself. Yet another kind of Transformer is a Repeater. The repeater is different because it can accept more than one OutputConnection: two, in fact. The instructions for a Repeater say:

1. Read the input.
2. Write this to one OutputConnection.
3. Write this to the other OutputConnection.

And, of course, the instructions for a (simple) PigLatin should say

1. Read the input.
2. Produce a new phrase containing all but the first letter, then the first letter, then the letters "ay".

3. Write this as output.

As you can see, the basic instructions for a Transformer are of the form

1. Read the input.
2. Produce a transformed version of it.
3. Write this as output.

We will begin by looking at the second of these instructions.

Strings

In Java, there is a special kind of object, called a String, that is designed to represent these words or phrases. In fact, in Java a String can be almost any sequence of characters typed between two double-quote marks, including spaces and most of the funny characters on your keyboard. (The double quotes aren't actually a part of the String itself; they simply indicate where it begins and ends.) For example, legitimate Java Strings include "Hello" and "this is a String" and even "&())__)&^%^^". (Strings don't have to make sense.) The Transformers that we will build are really StringTransformers, since each one takes in a String at a time and produces a corresponding, potentially new or transformed String as output.

String Concatenation

Once you have a String, there are several things that you can do with it. For example, you can use two Strings to produce a third (new) String using the String concatenation operator, +. In Java,

```
"this is a String" + "%%^$^&&)) mumble blatz"
```

is for all intents and purposes the same as just typing the single String

```
"this is a String%%^$^&&)) mumble blatz"
```

[Footnote: Note that there is no space between the g at the end of String and the % at the beginning of %%^$^&&))] So, for example, a NameDropper transformer might use + to create a new String using the input it reads, the name of the particular dropper, and the word "says". Pig Latin and Ubby Dubby might use +, too, but they'll have to pull apart the String they read in first.

String Methods

Java Strings are actually rather sophisticated objects. Not only can you do things with them, they can do things with themselves. For example, you can ask the String "Hello" to give you a new String that has all of the same letters in the same order, but uses only upper case letters. (This would produce "HELLO".) The way that the String does this is called a **method**, and you ask the String to do this by **invoking** its method. In this case, the name of the method that each String has is `toUpperCase()`. You ask the String to give you its upper-case-equivalent by putting a `.` after the String, then its method name:

```
"Hello".toUpperCase()
```

yields the same thing as `"HELLO"`.

You can also ask a String for a substring of itself. In a String, each character is numbered, starting with 0. (That is, the 0th character in "Hello" is the H; the o is the 4th character.)[Footnote: Computer scientists almost always number things from 0. This is apparently an occupational hazard.] So you can specify the substring that you want You can do this by supplying the index of the first character of the substring, or by supplying the indices of the first and last characters. `"Hello".substring(3)` is `"lo"`; `"Hello".substring(1,3)` is `"ell"`; and `"Hello".substring(0)` is still `"Hello"`.

These and other useful functions are summarized in the sidebar on String Methods.

Selected String Methods

Below are some selected methods that can be invoked on individual Strings, along with brief explanations and examples of their usage.

- `toUpperCase()` produces a String just like the String you start with, but in which all letters are capitalized. For example,
  `"MixedCaseString".toUpperCase()`
  produces
  `"MIXEDCASESTRING"`
- `toLowerCase()` produces a similar String in which all letters are in lower case. So
  `"MixedCaseString".toLowerCase()`
  produces
  `"mixedcasestring"`
- `trim()` produces a similar String in which all leading and trailing white space (spaces, tabs, etc.) has been removed. So
  `"    a very spacey String    ".trim()`
  is just
  `"a very spacey String"`
- `substring( fromIndex )` produces a shorter String containing the same characters that you started with, but beginning at index *`fromIndex`*. Bear in mind that the index of the first character of a String is 0.
  `substring( fromIndex, toIndex )` produces the substring that begins at index *`fromIndex`* and ends at *`toIndex`*` - 1`.
  `"Hello".substring(3)` is `"lo"`
  `"Hello".substring(1,4)` is `"ell"`, and
  `"Hello".substring(0)` is `"Hello"` again.
- `length()` returns the number of characters in the String. For example,
  `"Tee hee!".length()`
  is 8. Since the String is indexed starting at 0, the index of the final character in the String is the String's `length() - 1`.
- `replace( old, new )` requires two characters, *`old`* and *`new`*, and produces a new String in which each occurrence of *`old`* is replaced by *`new`*: [Footnote: A character is, roughly, a single

alphanumeric or symbolic character (one keystroke) inside single quotation marks. For more detail on what exactly constitutes a character, see the chapter on Java types.] For example,

```
"Tee hee!".replace('e', '*' )
```

produces

```
"T** h**!"
```

- `charAt( pos )` requires an index into the String and returns the character at that index. Recall that Strings are indexed starting at 0.
  `"Hello".charAt( 2 )` is the same as `"Hello".charAt( 3 )`
- `indexOf( character )` returns the lowest number that is an index of *`character`* in the String.
  `"Hello".indexOf( 'H' )` is 0 and
  `"Hello".indexOf( 'l' )` is 2. Also,
  `"Hello".indexOf( 'x' )` is -1, indicating that 'x' does not appear in "Hello".
- `lastIndexOf( character )` returns the highest number that is an index of *`character`* in the String.
  `"Hello".lastIndexOf( 'H' )` is 0 and
  `"Hello".lastIndexOf( 'x' )` is -1, but
  `"Hello".lastIndexOf( 'l' )` is 3.

Rules and Methods

Using the String manipulations described in the previous section and sidebar, we can construct the instructions that a variety of Transformers would use to transform a String. For example, we might write:

to `transform` a `String` (say, `thePhrase`),

```
return thePhrase.toUpperCase();
```

This rule describes the transformation rule for an UpperCaser. Note that theString is intended to stand in for whatever String needs to be transformed. The transformation rule can't operate unless you give it a String. Within the body of the transformation rule, a temporary name (in this case, thePhrase) is used to refer to this supplied String. The formal term for such a piece of supplied information is an **argument**, and the formal term for the temporary name that is used to refer to it is a **parameter**.

A different transformation rule -- this one for a pedantic Transformer that seems to think it knows everything -- might say

to `transform` a `String` (say, `whatToSay`),

```
return "Obviously " + whatToSay;
```

Note that we have chosen a different temporary name to represent the String argument. The parameter name doesn't matter; we can choose whatever (legal Java) name we wish.[Footnote: Legal Java names are covered in the sidebar on Java names in the chapter on Types.] It can be the same name in every transformer rule, or different in each one. It is only important that we use the same name in a particular rule both when we're specifying the parameter (in the first line of the rule) and in the body of the rule.

Q. Can you think of another kind of Transformer and write its rule? Remember, it should take a String and produce a String.

The rules as we've presented them aren't really Java code, but they are pretty close. To make them legal Java, we need to add a bit more formality and syntax. The formal name for a rule in Java is a **method**, just like the String methods -- `toUpperCase()`, `substring( index )`, etc. -- above. Somewhere, someone has provided instructions for how to `toUpperCase()` so that you can use that method without worrying how it is done. Here, we are providing the instructions for `transform`, so that someone else can use it.

A definition of UpperCaser's transform method might say:

```
String transform ( String thePhrase )
{
    return thePhrase.toUpperCase();
   }
```

Aside from the syntax (the details of which are covered in chapters 6 and 7), the one big difference from the rule specification above is that the method definition begins with the word String to indicate that the method will produce a String when it is invoked.

Q. Quick quiz: How would you write the pedantic Transformer's transform method?

Classes and Instances

What we just described was how to specify a rule. This rule is the rule used by all Transformers of that particular type. In fact, the rule is really the only thing that distinguishes Transformers of that type from other Transformers. We can describe a type of Transformer by wrapping the method definition in a bit of code that says it's a type. In Java, a type that provides instructions implementing behavior is called a **class**.

```
class UpperCaser extends StringTransformer
{

     String transform ( String thePhrase )

     {
         return thePhrase.toUpperCase();
           }

}
```

This says that UpperCaser is a type (or class) that is very much like the more general class StringTransformer. Its behavior differs from generic StringTransformers by using the particular transform rule contained inside the braces {} that delineate UpperCaser's body.

Pedant is similar:

```
class Pedant extends StringTransformer
{
```

```
        String transform ( String whatToSay )

        {
            return "Obviously " + whatToSay;
              }

    }
```

Q. A class that uses your transformer rule should be very much like these. Can you write it?

These classes are descriptions of what an UpperCaser or a Pedant should do. They are not UpperCasers or Pedants themselves, though. They're really more like recipes from which a particular UpperCaser or a particular Pedant can be made. To make an UpperCaser, you use the special Java construction expression `new UpperCaser()`. Thisk "cooks up" a particular UpperCaser using the recipe we just wrote. A Pedant is created similarly, but using a different recipe: `new Pedant()`. If we say it again, we can "cook up" another Pedant: `new Pedant()`.

Stepping back, this is exactly what we want the buttons on our control panel to do. Pressing the button marked Pedantic Transformer should invoke the expression `new Pedant()`, causing an Pedant to appear on our screen. Pressing it again should invoke it again, making a second Pedant appear. We can connect these two together using other user interface functions. Now, if we send the String "I'm here!" through a Connector to the first Pedant, it should send the String "Obviously I'm here" to the second Pedant, and the second Pedant should produce "Obviously Obviously I'm here".

Q. Connecting a Pedant's output to an UpperCaser's input and supplying the Pedant with "not much" will produce "OBVIOUSLY NOT MUCH". What happens if you connect an UpperCaser's output to a Pedant's input?

Q. How about Pedant, then Pedant, then UpperCaser, then Pedant? Then UpperCaser?

Fields and Customized Parts

You can already see from the examples in the previous subsection how one class, or type, can describe many different instances. For example, phrases passed through the first Pedant contain at least one "Obviously" at the beginning; phrases passed through the second Pedant will begin with at least two "Obviously"s. But to really appreciate the power of multiple distinct instances of a type, we need to look at a type that has local state associated with each instance. The NameDropper Transformer type is a good example of this.

The transformation rule for NameDropper is

> to `transform` a `String` (say, `thePhrase`),
> `return` *my name* `+ " says " + thePhrase;`

But *my name* here isn't a parameter. It isn't a piece of information that is supplied to the NameDropper each time the NameDropper performs a transformation, the way that `thePhrase` is. Instead, *my name* is a persistent part of the NameDropper. And it is a part of the particular NameDropper instance, not a part of the NameDropper type. After all, each NameDropper drops its own name.

So where does this name come from? As each individual NameDropper is created, it must be supplied with a name. Then, the particular NameDropper remembers its own name, and when it comes time to `transform` a `String`, the NameDropper uses its own name.

To do this, we need to create a local storage spot that sticks around between transformations. This is done using a special kind of name that is associated with the NameDropper instance. Such a name is called a **field**. In this case, we'll use a field called `name`, because that's what it will hold. To make it clear in our code that we're referring to a field, we use a syntax sort-of like saying *my name*; we refer to the field using `this.name`. In Java, `this` is a way of letting an individual instance say "my own".

So the actual transform method for NameDropper should read:

```
String transform ( String thePhrase )
{
    return this.name + " says " + thePhrase;
      }
```

This way, if one NameDropper has the name Pat and another has the name Chris, Pat would transform the String "Hello" into "Pat says Hello" while Chris would make it "Chris says Hello".

[Picture of Pat and Chris transforming the same String, with field visible.]

This method definition needs to be embedded in a class, of course. We also need to add a bit more machinery to the class to make sure that the name is available when `transform` needs it. The first change is to actually create a place to put the name; the second is to write explicit instructions as to how to create a NameDropper so that it has a name from the very beginning. This second -- constructor -- rule will need to say:

to construct a `NameDropper` with a `String` (say, `whatTheNameShouldBe`),
assign *my name* the value of `whatTheNameShouldBe;`

When we translate this into Java using the special syntax for a constructor rule, it looks like this:

```
NameDropper( whatMyNameShouldBe )
{
    this.name = whatMyNameShouldBe;
}
```

So the whole NameDropper class reads:

```
class NameDropper extends StringTransformer
{

     String name;        // the persistent storage,
                         // a permanent part of each NameDropper

     NameDropper( whatMyNameShouldBe )
     {                                    // the creation rule
               this.name = whatMyNameShouldBe;
     }
```

```
        String transform ( String whatToSay )
        {                                   // the transform rule
                    return this.name + " says " + whatToSay;
              }

    }
```

Now, when we invoke NameDropper's construction method, we give it a parameter: `new NameDropper( "Pat" )`, for example.

We have actually seen -- or at least alluded to -- a similar situation earlier. When discussing the other entities that together constitute a Transformer, we said that the input-connection-acceptor's job was to stick the input connection it receives somewhere where the rest of the Transformer community can use it. Like NameDropper, the generic StringTransformer accomplishes this using a field.

Fields, methods, and constructors are the building blocks of Java objects. We will see each of these things in action in the next several chapters. In chapter 7, on Classes and Instances, we will go through each of these items in greater detail For now, it is enough to have a general sense of how things fit together.

Generality of the approach

In writing this code, we have relied on the existence of a generic StringTransformer class. In that class, we include rules for how to accept an input connection (using a field to store it away), how to accept an output connection, and how to create an individual StringTransformer, including creating its own instruction follower to explicitly invoke the transform method over and over again on each String read from the stored input connection. The ways in which this StringTransformer class is put together are much like the ways in which the examples here are constructed, but the StringTransformer class is about four times the size of the classes described above. The complete code for StringTransformer is included in the on-line supplement to this book.

The transformers that we have written here each obey the same general rules and interfaces. Each defines a transform method that takes a String and returns a String. The apparent uniformity among StringTransformers makes it possible for the connection mechanism that we outlined in the previous section to work with each of them. The differences among StringTransformer behaviors are hidden inside the transform method that each of them implements. In the course of this book, we will see many different cases in which hiding behavior behind a common interface makes a system more general and more powerful. Good design specifications are crucial; they amount to deciding in advance how entities will interact.

Summary

In this chapter, you have been exposed to many of the most basic pieces of Java programming. None of these has been presented in sufficient detail to achieve mastery of it. Each of these topics will be revisited, most in the next part of the book. But the example described above gives a context within which to place the detail that occupies the next several chapters.

In the next chapter, we will explore the role of types in Java systems and the relationship between types and names. The final chapter of this section looks at interfaces, the contracts that one type of object makes with another. In the next section, we turn to expressions -- such as method invocation, field access, instance construction, and even String concatenation -- and learn how evaluating an expression produces a value of a specified type. Expressions are combined to make statements, the step-by-step instructions of Java code that produce behavior and flow of control. Classes allow us to implement behavior and to encapsulate both instructions and local state -- such as the NameDropper's name -- into individual objects. And self-animating objects contain their own instruction followers that execute sequences of instructions over and over, communicating with other objects and interacting to provide desired behavior on an ongoing basis.

Suggested Problems

See the text for things marked with a **Q**. Also:

1. Implement LowerCaser.

2. Implement SentenceCaser (1st letter capitalized, rest not).

3. Implement Pig Latin.

4. An improved Pig Latin would leave the first letter in place if it were a vowel, and add -way instead. This requires understanding basic conditionals and flow of control. (See Statements.)

5. Ubby Dubby is pretty hard. You may want to look carefully at the chapter on Dispatch.

6. Combiners and Repeaters involve extending StringTransformer in other ways, overriding acceptInputConnection or acceptOutputConnection. (See the online code supplement for StringTransformer source code.)

7. Really challenging problem: extract words, one word at a time, only reading an input when all words have been used up.

This chapter is excerpted from a draft of *Introduction to Interactive Programming In Java*, a forthcoming textbook. It is a part of the course materials developed as a part of Lynn Andrea Stein's Rethinking CS101 Project at the Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>


WHILE (TRUE) {
ECHO
}