Algorithms in Java: Euclid's Algorithm

Euclid

Euclid (*fl.* 300 BCE) was a prominent Greek mathematician – generally considered to be the "Father of Geometry" – who wrote and taught at the Library of Alexandria during the reign of Ptolemy I. His most famous work, *Elements*, is arguably the most important mathematics textbook ever written.

In *Elements*, Euclid primarily writes about geometry[1], but he addresses number theory as well. In fact, the problem we'll work on in this activity was initially expressed in geometric terms by Euclid, but his solution is an important development in number theory.

Consider two line segments, of different lengths. Can we find a third line segment of such a length that this new line segment will evenly measure the other two (i.e. it will fit into each of the other two an integral number of times, with no portion left over)? What is the largest such line segment? Another way of expressing the problem is probably more familiar to you: Given two numbers, what is the greatest common divisor of the two? This is the problem solved by Euclid's algorithm.

The algorithm can be applied to various different kinds of numbers and algebraic quantities, but the most common examples use positive integers; we'll focus on those for our activity.

Euclid's Algorithm

Consider two positive integers, a and b. To find the GCD of these two, we follow these steps:

- While $(b \neq 0)$,
  - If $(a > b)$, then
    - Compute $(a - b)$, and set a equal to the result.

    Otherwise,
    - Compute $(b - a)$, and set b equal to the result.
- The current value of a is the GCD of the original a and b values.

This is close to what we would consider a *pseudocode*[2] expression of the algorithm, though it's more verbose than most pseudocode. In such expressions, the indentation is significant. For example, the lines underneath and indented to the right of "While $(b \neq 0)$" should be repeated until the condition "$(b \neq 0)$" is no longer true.

---

1 Until the 19th century CE, the geometry described in Euclid's *Elements* was generally considered to be the only geometry possible – or at least, the only geometry with any real world application. Since then, however, two distinct and very important non-Euclidean geometries have been formulated, which differ from Euclidean geometry on just one of Euclid's postulates.

2 Pseudocode is a high-level description of an algorithm, usually intended to be a human-readable, unambiguous stepping stone to its implementation. Pseudocode generally employs structural conventions common to many programming languages, but mostly avoids language-specific syntax.

Implementation

There are a number of ways to implement Euclid's algorithm, in virtually any programming language. We'll use Java for our implementation, and we'll translate our code almost directly from the description of the algorithm we just read.

In Java, almost all executable code is written in *classes*. A class is generally one of three types, or a combination thereof:

1. A structure used for declaring and defining a certain kind of object, and containing all of the attributes (variables) and behaviors (methods) for that kind of object. For example, the standard Java library has a `Window` class, which includes many of the basic attributes and behaviors common to almost all types of windows in a computer program. To create and display a window on the screen, we could create an object of the `Window` class (more often, we will create an object of a *subclass* of `Window` – i.e. a class that inherits all of the behaviors and attributes from `Window`, but adds some additional and/or more specialized functionality).

2. A collection of methods, not necessarily associated with a particular kind of object, but rather with a particular area of functionality. The standard `Math` class is a good example: we don't use it to create `Math` objects; instead, we use its methods to perform advanced mathematical computations (trigonometric functions, logarithmic functions, etc.) on the standard numeric types of data.

3. An *entry point* to a Java program, applet, or other package of functionality. For example, when we use an Internet browser, and navigate to a page that includes an applet, the main class for that applet includes a number of methods, with very specific *signatures* (names and expected parameter types) that are invoked by the browser, to start and stop the applet. If the main class of the applet didn't include those methods, with those specific signatures, the browser couldn't tell the applet to start running. Java programs have a similar requirement: even though they don't run inside a browser window, they do need a main class, with a `main` method (with a specific signature), so that the Java Virtual Machine (JVM) knows where the program starts.

In this activity, we'll build one class of the second type (i.e. a library of coherent functionality – in this case, the only capability of the library will be to find the GCD of two positive integers), and another class of the third type.

Many testing tasks in this activity assume that you are using the DrJava development environment (which can be downloaded from http://www.drjava.org). However, the programming itself can be done with virtually any IDE or text editor, as long as a Java Development Kit (JDK) is installed.

The `Euclid` Class

In your Java programming environment, create a new source file, if a new empty file is not already displayed. (If you're using NetBeans or another IDE that requires you to give the name of the class when you first create it, call it "**`Euclid`**").

In computer programming, we often start a new program by building something very simple – so simple that it might do nothing at all – *but it does it correctly*. (This may sound like nonsense, but just remember: even for experienced programmers, it's better to start with an error-free piece of code, and add to it – and test it – in small increments, than it is to write a lot of code at once, and then try to go back and correct the problems that almost invariably creep in.) So, to start, let's create a class that does nothing:

```
public class Euclid {

}
```

Pay very close attention to spelling and capitalization: Java, like C, C++, C#, and many other languages, is case-sensitive (i.e. capitalization matters); virtually all programming languages are very literal-minded when it comes to spelling – close doesn't count. Also, the left curly brace after the class name, and the matching right curly brace, are required (though the left curly brace doesn't need to be on the same line as the class declaration); all of the variables (attributes) and methods (behaviors) that we define for this class must be enclosed within that pair of braces.

If we've spelled everything correctly, we should be able to save and compile our file now. When saving, be sure to save the file using the exact same name and capitalization that you used in the class name (**`Euclid`**), but with the **`.java`** file extension. So this file must be saved with the name **`Euclid.java`**. (If you're using DrJava, the default file name in the **Save** dialog is generated automatically – and correctly – from the class name; be sure not to change this default name.)

What does our class do? Nothing yet. It's a public class (i.e. it can be seen and used by other Java code outside this file) called **`Euclid`**, but without any attributes or behaviors.

Since our class won't be used to create objects with their own attributes, but to compute the GCD of two numbers, we probably don't need to declare any class variables. Instead, let's go ahead and start creating the method that will (eventually) implement Euclid's algorithm:

```
public class Euclid {

    public static int getGcd(int a, int b) {
        return a;
    }

}
```

(In this document, the new code added at each step will be shown with a lighter background, in bold, italic, blue type. Also, notice that DrJava – like many other Java development environments – automatically indents the statements following an opening curly

brace, and then out-dents the closing curly brace, so that it is indented to the same level as the line where the opening curly brace appears.)

Save and compile your code again. If it compiles without error, you can be confident that it's *syntactically correct* – that is, you're correctly following the rules of Java syntax. However, this doesn't mean your code is *logically correct* – i.e. that it does the job it's intended to do. In fact, as we will see, it isn't (yet) logically correct, but that's fine – we're not done yet!

Let's look at what we just added: We created a public method (i.e. it can be called from outside this class). It's also *static* – this means that the method can be called without first having to create a `Euclid` object based on the class. For a result, the method returns an `int`, which is the basic integer data type in Java. The name of the method is `getGcd`; this follows the suggested convention of starting method names with verbs, since they perform actions. The method takes two `int` values as parameters; these will be referred to in the method as `a` and `b`, to match the articulation of Euclid's algorithm we read earlier (we don't have to make the parameter names match those names, but we might as well do so, since we don't really have more meaningful names to use for the parameters).

Right now, the `getGcd` method simply returns the value of `a` as a result, without first performing the steps required to find the GCD. Clearly, this isn't logically correct, but we'll fix that right now – by writing the code to implement the algorithm we saw earlier, and inserting it between the curly braces of the `getGcd` method. The logic expressed in the code should look familiar to you after reading the algorithm, even though the programming language is new to you.

```
public class Euclid {

    public static int getGcd(int a, int b) {
        while (b != 0) {
            if (a > b) {
                a -= b;
            }
            else {
                b -= a;
            }
        }
        return a;
    }

}
```

Some explanation would be useful here. First, notice that the way we write the logical condition "$b \neq 0$" in Java is "`b != 0`"; the exclamation point means "not". Next, notice the way we subtract `b` from `a`, and put the result back into `a`: "`a -= b`" This an example of the use of a *compound assignment operator*. In this case, the operator is "`-=`", and it means "subtract the value on the right from the value of the variable on the left, and put the result into the variable on the left." We could also have written "`a = a - b`", and the result would have been the same; however, the shorter form is often executed in a slightly more efficient fashion. (There are also compound assignment operators for addition, multiplication, division, and other common operations.)

The code includes two important Java program flow structures:

- **`while`**

  This keyword is used to execute some statement repeatedly, as long as a specified condition is true. **`while`** is followed by a boolean (i.e. true/false) condition expression, which must be in parentheses, and then a statement (which may be a simple statement, ending in a semicolon, or a statement block, enclosed in curly braces). The condition is tested, and if the result is true, the statement is executed. This process is repeated until the condition is no longer true.

- **`if-else`**

  Like **`while`**, **`if`** is used to execute a statement conditionally (also like **`while`**, the boolean condition must follow the **`if`** keyword, in parentheses, and the statement must follow the condition). Unlike **`while`**, however, this does not happen repeatedly, but only once (of course, **`if`** could be used in a statement executed by **`while`**, in which case it might be executed repeatedly; however, this repeated execution is due to the **`while`** keyword, and not **`if`**). On the other hand, **`if`** has a feature that **`while`** doesn't have: when followed by **`else`**, the statement after **`else`** will be executed when the specified condition is false.

The **`return`** statement we previously wrote at the end of the method now makes more sense: by following the steps of the algorithm, **`a`** will contain the correct GCD value when the condition "**`b != 0`**" is no longer true – i.e. when **`b`** is equal to zero.

Testing the Euclid Class

If you're using the DrJava environment to build your code, testing the Euclid class is very straightforward.

First, save and compile the class, and fix any problems reported by the compiler. (The great majority of compilation problems, even for experienced programmers, are caused by forgotten semi-colons, inconsistent capitalization, and misspellings.)

Next, click on the **Interactions** tab (usually located in the lower part of the DrJava window). This is a very handy tool for interactively executing Java statements, without having to build and run a complete Java program. Let's start by typing a few simple examples. Type the following lines, one at a time, and see what happens. Note that the "**>**" character is what the **Interactions** pane displays as a prompt; don't type that part of the line. Also, lines that don't start with "**>**" are values displayed by DrJava (in response to what was typed in previous lines), and are shown with a darker background here; don't type those lines.

```
> int x = 7;
> int y = x * 25;
> y
175
>
```

Did you see the same result? Did you see any error messages? If so, you will probably need to check your typing.

What did we do? First, we declared an integer variable called `x`, with a value of `7`. Next, we declared another integer variable, `y` this time, with a value of `25 * x`. (As you've probably figured out, or might have known already, most programming languages use the asterisk character, "`*`", as the symbol for multiplication. So here, we are multiplying 25 by the value of `x`, and assigning the result to `y`.) Finally, we typed the variable name `y` without an ending semicolon; that isn't valid Java syntax, but it's how we tell the DrJava **Interactions** pane to display the value of an expression – in this case, we asked it to display the value of the variable `y`, which was 175 (i.e. 25×7).

Now, let's try it with the class we created:

```
> int a = 27;
> int b = 45;
> Euclid.getGcd(a, b)
9
>
```

The first few lines aren't much different than what we typed before. But the third line is new: here, we're calling the method we just wrote (in the Euclid class), which finds the GCD of the `a` and `b` – in this case, of `27` and `45`. Did you get a different result? Is 9 indeed the GCD of `27` and `45`?

Here, we called our two numbers `a` and `b`, just as we did the method itself. However, when we call a method, we aren't limited to using the same variable names used by the method. Type the following lines in the Interactions pane, and this should become clearer:

```
> int x = 187;
> int y = 121;
> Euclid.getGcd(x, y)
11
> Euclid.getGcd(x - 2, 333)
37
> Euclid.getGcd(5115, 1581)
93
>
```

It should now be apparent that just because our static `getGcd` method has parameters that it refers to as `a` and `b`, that doesn't mean we have to pass values of variables by the same names to that parameter. In fact, we can pass literal numerical values, other variables, or computed expressions – as long as what we pass matches the expected data type.

There's another important Java detail about passing parameters that we should know: when we have a method that takes parameters of the simple, intrinsic Java data types (e.g. `int`), and not arrays or objects (more about arrays later), any changes we make to those parameters inside our method don't affect the values of variables passed to the method. For example, even though the `getGcd` method modifies the values its parameters, the values of `x` and `y` in the above **Interactions** code remain unchanged, even after using them in the call to `Euclid.getGcd`.

Protecting Our Class Against Bad Data

There's one more addition we should make to our `Euclid` class. While Euclid's algorithm can be written to work with many types of numbers (real numbers, complex numbers – even polynomials), the way we've written it limits it to handling only positive integers. However, we haven't prevented the methods of our class from being called with negative integers or zero; if we used values like that, our method wouldn't behave very well (go ahead and try it – *but be sure to save any changes you've made first*; you might end up having to restart DrJava – or at least, you may need to right-click or Ctrl-click in the **Interactions** pane, and select **Reset Interactions** from the context menu that appears.).

It's always tempting to say: "I'll just be careful not to use invalid values in my code, and avoid the problem that way." But imagine that one student has written the `Euclid` class, and another (a classmate of the first, maybe) wants to use that class in his own program, which needs to compute and display GCDs. However, the second person doesn't know that only positive integers should be used (the first person forgot to pass that vital information along), and gets very frustrated when his program keeps getting stuck in infinite loops! It would be much better if the `getGcd` method had a way of sending an error – more formally called *throwing an exception* – back to the code that tried to pass invalid values to it. (Though handling thrown exceptions in general is beyond the scope of this activity, we'll see a simple example of this later.)

Fortunately, there's already a type of exception in the standard Java library that fits our needs well: `IllegalArgumentException`. Let's change our code, so that if one of the values passed to `getGcd` is less than or equal to zero, an `IllegalArgumentException` will be thrown. We'll also generate an informative message, so that any programmer who might use our class for computing GCDs, and who unknowingly tries to use it for negative numbers or zero, will know what he or she did wrong.

Pay attention to the highlighting in this set of changes, because there are actually two places where you need to add code: just after the start of the class, and inside the `getGcd` method.

```
public class Euclid {

    private static final String EXCEPTION_MSG =
        "Invalid value (%d); only positive integers are allowed.";

    public static int getGcd(int a, int b) {
        if (a < 0) {
            throw new IllegalArgumentException(
                String.format(EXCEPTION_MSG, a));
        }
        else if (b < 0) {
            throw new IllegalArgumentException(
                String.format(EXCEPTION_MSG, b));
        }
        while (b != 0) {
            if (a > b) {
                a -= b;
            }
            else {
                b -= a;
            }
        }
        return a;
    }

}
```

(Be careful to make sure that you have balanced parentheses in the calls to the `IllegalArgumentException` constructor. Also, note that the line breaks in the new lines shown above are not required; however, it is generally a good idea to avoid very long lines in code.)

The first change includes a new variable modifier: `final`. This is how we tell Java that the initial value we're assigning to a variable is also its final value. If our code later assigned another value to such a variable, the Java compiler would refuse to compile the code, since it would violate the `final` aspect of the variable declaration. We might say that a `static final` variable isn't really a variable, but rather a constant that will hold a single value as long as the class is in scope – i.e. as long as a program using the class is running. In this case, we're declaring `EXCEPTION_MSG` as a constant `String` (a type of object that holds a sequence of characters); this string has a placeholder in it, which is identified by the "`%`" symbol, followed by a formatting code.

To understand how the placeholder in this string is used, let's look at the code where we refer to `EXCEPTION_MSG`. With the expression "`String.format(EXCEPTION_MSG, a)`", we're calling the `format` method of the `String` class. That method formats a string of characters by replacing any placeholders with the values passed in any additional parameters to the method, and formatting those values according to the formatting and conversion codes following the placeholder character. In this case, we're replacing the placeholder with the value of `a`, and formatting that value as a decimal integer (that's what the "`d`" means; alternatively, we could use "`x`" or "`X`" to format an integer value as a hexadecimal value, or "`o`" to format it as an octal value.).

We're passing the formatted string of characters as a parameter value to the *constructor* for the `IllegalArgumentException` class (a constructor is a special method, used to create an instance of a class). We are then *throwing* that `IllegalArgumentException` object – i.e. sending it back to the method that called our `getGcd` method. If that calling method doesn't include statements to *catch* the exception, the exception is thrown to the method that called the method that called our method … and so on.

Let's try another test, to make things a bit clearer. First, save and compile your code, fixing any errors that are reported by the compiler until you can compile successfully. Next, type the following into the **Interactions** pane:

```
> Euclid.getGcd(185, 333)
37
> Euclid.getGcd(185, -333)
java.lang.IllegalArgumentException: Invalid value (-333); only positive
integers are allowed.
      at at Euclid.getGcd(Euclid.java:12)
>
```

What is this message telling us? First, it's saying that an `IllegalArgumentException` was thrown, and that it included the message: "`Invalid value (-333); only positive integers are allowed.`" (Of course, this is the message produced in our code.) Next, it's telling us that the `IllegalArgumentException` object was thrown from the `getGcd` method of the `Euclid` class, on line 12 of the file `Euclid.java`.

If we were to write a GUI program that let the user type in integer values, and then computed and displayed the GCD for those values, that GUI wouldn't even have to know that only positive integers are allowed. Instead, it could simply catch any exception thrown by the methods in the `Euclid` class, display the message (e.g. "Invalid value (-333); only positive integers are allowed.") to the user, and have the user type in new values. Of course, this would be a pretty primitive user interface – *but invalid inputs wouldn't make it crash!*

Building a GUI

Remember the last page, when we talked about building a GUI that would use the `Euclid` class to compute the GCD of pairs of positive integers? Well, now we'll do exactly that.

This will be a very primitive user interface. Also, the code will have much less explanation than our `Euclid` class. The reason for this is that learning how to use the Swing toolkit (the set of Java classes used for building GUIs) is outside the scope of this activity. However, we should learn at least a little bit about the classes we'll use in this task. Each of these new classes is listed below with its *fully qualified* name, which includes the name of the *package* containing the class, and the name of the class itself. These fully qualified names are used in the `import` statements in the code, where we tell the Java compiler which classes we'll be using (besides those located in the current package, which we can use without importing).

- `java.awt.event.ActionListener`

  An `ActionListener` object is one with an `actionPerformed` method, with a specific signature. If an `ActionListener` instance is attached to a type of event on a component, and that event occurs, the `actionPerformed` method in the `ActionListener` instance will be called. In this case, the only action we'll be listening for is the user clicking on a button; when that happens, we'll use our `Euclid` class to compute the GCD, and then display the result in the GUI window.

- `java.awt.event.ActionEvent`

  Objects of this type contain information about an event (e.g. a button click) in a GUI application.

- `javax.swing.JButton`

  A `JButton` object is pretty much exactly what it sounds like: a button. It can be displayed an image, as text, or both. When the user presses the button, the `actionPerformed` method of `ActionListener` objects (there can be more than one) associated with the button are called.

- javax.swing.JLabel

  Sometimes, rather than display a field that the user can edit, we just want to display static text. Text like this is commonly used for field prompts and labels, instruction and other informational text, etc. JLabel objects are used for displaying this kind of test.

- `javax.swing.JOptionPane`

  The `JOptionPane` class is very useful for displaying simple dialog windows and user prompts. Since our user interface will be very simple, we'll implement it with an dialog window constructed by this class.

- `javax.swing.JTextField`

  This type of object is used for editable text inputs. We'll use to of these objects to let the user provide the input numbers, and a third (which we won't let the user edit) for displaying the GCD of those two numbers.

The "Do-Almost-Nothing" Class

Once again, start by creating a new class file, located in the same directory as `Euclid.java`. This will be the `EuclidGui` class, and we'll get it started with the following code:

```
public class EuclidGui {

    public static void main(String[] args) {
    }

}
```

Save this code (as `EuclidGui.java`) and compile. As with the first version of our `Euclid` class, this new class is very minimal at the moment. It does have one method – which does nothing. However, this `main` method is very important for our class. `EuclidGui` is one of those entry point classes referred to early in this activity, and the `main` method is the key. When the Java Virtual Machine (JVM) launches a Java program, that program may have dozens – even hundreds – of classes. But one class will be designated as the main class, and that class must have a `main` method with the signature seen above: it must be public (visible outside the class – otherwise, the JVM wouldn't be able to see it!); it must be static; it must not return a value; it must be named "`main`"; it must take as a parameter an array of `String` objects (the square brackets are used to declare and use an array; we'll see another array in the next part of the activity, but learning how to use them is beyond the scope of this activity).

Now that we have a `main` method with the expected signature, we can actually run this class as a Java program. What do you think will happen if we do? What does the `main` method do? Try it and see if your prediction is right. (In DrJava, you can run the current class as a program with the **Run** button in the toolbar, or with the **Tools/Run Document's Main Method** menu selection.)

Create and Displaying the Swing Components

Now let's declare the variables that we'll use for the different UI components (fields, labels, buttons), and write the code to initialize those variables and display the components. For the sake of simplicity, we'll put all of the code to initialize and display the UI components in the `main` method; in a real-world program, we'd generally create one or more additional methods for this purpose.

The Swing classes are all in different packages than the one in which we've compiled our two classes, so we need to tell Java where it can find those classes. We do that with `import` statements at the start of a Java file; we'll need a few of those here. We'll also create some constants, to hold the literal text values that will be displayed by the UI.

Once again, be careful when reading these additions to the code: there are lines that must be added at the top of the file (before the declaration of the class itself), and lines that need to be inserted inside the `main` method.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JTextField;

public class EuclidGui {

    private static final String PROMPT_A = "#1";
    private static final String PROMPT_B = "#2";
    private static final String BUTTON_TEXT = "Get GCD >>";
    private static final String EXCEPTION_TITLE = "Input Exception";
    private static final String INSTRUCTIONS =
            "Type two integers and press 'Get GCD'";
    private static final String DIALOG_TITLE = "Euclid's Algorithm";
    private static final int FIELD_WIDTH = 6;

    public static void main(String[] args) {
        final JTextField valueA = new JTextField(FIELD_WIDTH);
        final JTextField valueB = new JTextField(FIELD_WIDTH);
        final JTextField valueGcd = new JTextField(FIELD_WIDTH);
        JLabel labelA = new JLabel(PROMPT_A);
        JLabel labelB = new JLabel(PROMPT_B);
        JButton computeButton = new JButton(BUTTON_TEXT);
        Object[] options = new Object[] {labelA, valueA, labelB,
                valueB, computeButton, valueGcd};
        valueGcd.setEditable(false);
        computeButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                try {
                    int a = Integer.parseInt(valueA.getText());
                    int b = Integer.parseInt(valueB.getText());
                    int gcd = Euclid.getGcd(a, b);
                    valueGcd.setText(Integer.toString(gcd));
                }
                catch (Exception e) {
                    JOptionPane.showMessageDialog(null, e.getMessage(),
                        EXCEPTION_TITLE, JOptionPane.ERROR_MESSAGE);
                }
            }
        });
        JOptionPane.showOptionDialog(null, INSTRUCTIONS, DIALOG_TITLE,
                JOptionPane.YES_NO_OPTION, JOptionPane.PLAIN_MESSAGE,
                null, options, null);
    }

}
```

This time, we have several constant (i.e. `static final` variable) declarations. As a rule, it's a good idea to avoid putting lots of literal constant values directly in executable code; instead, most of these (with the exception of some trivial and immediately understood values, like the numbers 0 and 1, the empty string, etc.) should be assigned to constants at the start of a class. This makes the code more readable, and easier to maintain when changes are needed.

Our `main` method also has a number of *local variables* – i.e. variables declared inside a method, or inside a pair of curly braces that make up a statement block. These variables are only defined within the method or statement block where they declared, and their values disappear once the method or statement block completes execution.

One of the local variables in the `main` method, `options`, is an array. Instead of holding a single value, an array holds multiple values of the same type (in this case, the type is the very general `Object` class; the contents of `options` can be `Object` instances, or instances of any class that is based directly or indirectly on `Object` – i.e. any Java object at all). As noted previously, an array variable can be recognized by square brackets, both in the declaration and the use of an array. Often, we specify the size of an array explicitly when it is created. Here, however, we're specifying the values the array will hold with a set of curly braces, and the size of the array is determined by the number of values; this is one kind of *array initializer* expression.

The trickiest part about the code is the call to the `computeButton.addActionListener` method. We use this method to associate an `ActionListener` object with a specific component (in this case, `computeButton`). It's a fairly common practice to do just what we did here, and create an `ActionListener` object "on the fly". In fact, because `ActionListener` isn't even a class, but is instead an interface (somewhat similar to a class, but one where any methods included are only declared, and not implemented – the actual code of the method is left out, and must be provided by the class that implements the interface), we're actually defining a class and creating an instance of that class right in our code, in the parameter values we're passing to `computeButton.addActionListener`. This part of Java programming is confusing, and takes practice. For now, just pay close attention to the curly braces and the parentheses: in code like this, it's can be very challenging to keep them balanced.

In the `actionPerformed` method (in the new `ActionListener` we created), you'll see code that catches exceptions. The `try-catch` keywords let us execute a statement (a simple statement, or a statement block in curly braces) and *catch* specified exceptions that might be thrown by that statement. In this case, we're catching any type of exception that might be thrown (including the `IllegalArgumentException` that our `Euclid.getGcd` method throws when negative values are passed to it), and handling them all the same way: presenting an alert dialog to the user.

One more thing to notice about the `actionPerformed` method, and its relationship to `main`: although the enclosing `ActionListener` is defined and instantiated in the `main` method, methods in an *anonymous inner class* (as our `ActionListener` is) don't have access to any of the enclosing method's local variables unless those variables are declared `final`. In this case, the `actionPerformed` method must be able to use `valueA`, `valueB`, and `valueGcd`, which are local variables in the `main` method. So we declare these variables `final`, to ensure sure that they're available to the `actionPerformed` method. (Don't worry if this is confusing now: with practice, it will become clear.)

Testing EuclidGui

Save and compile the `EuclidGui.java` file. When you've successfully compiled the class, run it. In some IDEs, this requires that you specify `EuclidGui` as your program's main class. In DrJava, simply click the **Run** button while `EuclidGui.java` is the active file in the editor.

When you run the `EuclidGui` class, you should see something like this:

See what happens when you type in two positive integers, and then press the **Get GCD** button. What happens when you type in a negative integer, or zero, in one of the boxes, and press **Get GCD**? Do you see the error message you programmed in the `Euclid` class? What happens if you type something that isn't an integer – non-numeric text, or a number with a value after the decimal point? Where do you think the resulting error message is coming from – i.e. what statement in the block statement following the `try` keyword is throwing that exception? Is it coming from code we wrote, or somewhere else?

Review

Congratulations! In this activity, you have accomplished the following:

- Created a class containing a method that implemented Euclid's algorithm for finding the GCD of two positive integers.
- Learned the basics of declaring method parameters, passing values for those parameters in method calls, and using those values in the method code.
- Learned how to use two important Java logical flow structures: `while` and `if-else`.
- Written code to detect invalid inputs to the algorithm, throwing a Java exception (with a meaningful error message) back to the caller in such cases.
- Built a Java program that implemented a Swing-based GUI, allowing a user to type in two positive integers.
- Implemented an action listener that invoked your method for computing the GCD, with data provided by the user.
- Included code in the action listener to catch thrown exceptions – not just the exception you programmed the `Euclid.getGcd` method to throw, but others as well.
- Displayed the returned GCD to the user in a read-only GUI field.