Java and the Art of Code

Burton Rosenberg

February 16, 2004

1 Message and Meaning

A program is a series of instructions to a machine capable of simple arithmetic and logical calculation. There are two aspects to a program: the program as written and the program as run. To understand programming is to study and master both aspects.

As text, a program must follow the programming language's grammar. The textual aspect of a program is also known as *syntax.* If a program is correct grammatically, it is often said to be *syntaticly correct.* The program might not "mean anything", that is, do anything useful, or correspond in action to what the programmer intended. That the programmer intends the program to do when run is, in a sense, the "meaning" of the program, and is often called the *semantics.* When a program runs correctly then it is said to be *semantically correct.*

Compared to the grammar of human languages, the grammar of computer languages are straightforward and very strict. A computer program called a *compiler* applies the rules of the grammar to the written program, and if the text obeys the rules, the compiler "understands" the program. It then rewrites the program in a language more suitable for the simple capabilities of the computer.

It is sometimes possible that a program runs incorrectly because the compiler understood the program in a different way than did the programmer writing the program. This is less likely to happen in Java than, say, in C or in C++. Their was a particular effort on the definers of the Java language to remove the possibility for such understandings. Certain features are lacking from Java because their inclusion necessarily opened the door to misunderstandings.

However, the more usual reason for a program to run incorrectly is that the what the programmer was misguided in his or her thinking, and by consequence, misguided the computer as well. For this sort of mistake, there is no cure inside of grammar. Programming disciplines are created to aid the programmer think correctly. These must be studied along with the syntax.

The program HelloWorld.java is given as an example of a Java program.

```
class HelloWorld {
    public static void main( String [ ] args ) {
      System.out.println("Hello World") ;
   }
}
```

Figure 1: HelloWorld.java

2 Message

2.1 Variables

The first fundamental concept of programming language is a *variable.* A variable is a named box in which a value can be stored and retrieved. (A variable can be thought of in other ways, but this is a good working model to begin with.)

The values which are stored in variables are classed according to *type.* The most common type is integer. Integer values are, for example, $0, 1, 2$ and so forth, including negative integers. Although there are an infinity of integers, most computer languages admit only a finite number of integer values, depending on the word size of the computer hardware. In Java the smallest integer is -2147483648 and the largest is 2147483647. Another Java type is *boolean.* It has two values, *true* and *false.* A variable of type boolean can store either true or false, and nothing else.

Some languages are less picky about types than others. Language which are picky are called *strongly typed languages*, the others are *weakly typed.* Java is a strongly typed language: a variable is declared with its name and its type. Only values of the variable's type can be stored in the variable.

An *identifier* must be chosen to name the variable. An identifier is a precise part of Java grammar. It is defined as any sequence of upper or lower case letters, numbers, or the special underscore character, which begins with a letter or the underscore. For instance, you can name your variable "u2" or "current_page_count". However, excluded from identifiers are certain reserved words called *keywords* and *literals.*

Keywords are words that the compiler locks onto in the deconstruction of the program, they signal the structure of the program and are not allowed to be used as variable names. A *literal* is a textual representation of a value. For instance, 1 is the textual representation of the integer value 1. It could not be confused with an identifier because it does not follow the rules for identifiers — it does not begin with a letter or an underscore. However *true* is also a literal, representing the value true of boolean type. It can be confused with an identifier but it is not an identifier. It is and literal and therefore excluded from being an identifier. You cannot name a variable "true".

Although you are legally free to name your variables with any identifier, there is a common culture which shapes the correct selection of names for your variables. It differs from language to language and employer to employer, but everywhere there is a naming discipline. The point of the naming

```
boolean b ;
int i, j, k ;
char c ;
```

Figure 2: Example declaration statements

discipline is to provide hints about the purpose, importance and interrelationships of the variables in the code.

One dimension of this discipline is the choice of meaningful names, or sometimes purposefully meaningless names. An important variable should be given a name descriptive of its purpose. A variable keeping track of page hits should be called *totalPageHits,* for instance. On the other hand, a short generic name, such as *scb2*, standing for "soda choice button two", is a valuable hint to the reader that this variable is itself somewhat generic and will be used in a paradigmatic and stereotypic manner.

The second dimension of this discipline is the capitalization pattern of the name. Customary Java variables begin with small letters. To begin a name with a capital letter strongly suggests a class, not a variable. A name all in capitals, with underscores to separate the words in the name, will be the name of a constant, such as *PI* or *MAX_USERS_ALLOWED.*

Variables are introduced by a *declaration statement*, a line in the Java program that brings together the identifier naming the variable with a keyword expressing the type of the variable. Typical keywords for types are *int* for integer, *boolean* for boolean, or *char* for characters. The *scope* of a variable is that region of the program text in which the variable name is known to the compiler. A declaration statement begins the variable's scope. We will return later to the question of the end of a variable's scope. Example declaration statements are given in figure 2.

2.2 Expressions

Most of what happens in a program is calculation. Not only numerical calculation, but logical calculation, text manipulation, database searching. Calculation is accomplished by the evaluation of expressions.

An *expression* is a combination of variables, literals, operators and method calls which render to a value. Expressions are built up out of other expressions. The simplest expressions are literals or variables, all by themselves. A literal as an expression evaluates to itself. The literal 3 evaluates to the value 3, a value of type integer. A variable as an expression evaluates (in most cases) to its currently stored value. If the variable named i is of integer type, and it currently stores the value 7, then i as an expression evaluates to 7.

Operators combine expressions to form new expressions. For instance,

```
i + 3
```

is an expression in which the plus operator combines a variable with a literal. The entire expression evaluates to a value. In order to do this, the addition operator must evaluate the expressions i and 3 for their values; and then combine these values rendering a value for the entire expression. If i evaluates to 7, then $i + 3$ evaluates to 10, since 3 evaluates to 3 and 7 plus 3 is 10.

Plus is a *binary operator*, that is, it takes two arguments. By conventions borrowed from standard mathematics, one argument stands on the operator's left hand side, the other on the operator's right hand side. There are also *unary operators*, taking one argument, and *ternary operators*, taking three arguments, although these are unusual. A common unary operator is negation. For instance, -1 might look like a literal, but it is an expression involving the unary negation operator. The entire expression evaluates to -1, of course, but by a two step process: first the literal 1 is evaluated, then the negation operation is applied to the resulting value.

The *assignment operator* important as well as curious. Consider a typical expression involving the assignment operator,

```
i = ( i + 3 )
```

Notice the parenthesis. Expressions are built up out of other expressions, typically connecting them with operators. To keep clear which arguments apply to each operator, parenthesis provide grouping. In this case, the right hand argument to the assignment operator is certainly $i + 3$, as the parenthesis make clear. However, in most cases parenthesis are not needed as the computer will read the expression even without parenthesis in the manner which we expect. There are strict rules for this but we will not stop now to consider them.

What is curious about the assignment operator is the manner in which it evaluates its left hand argument. It expects to find in that location a variable; and it does not evaluate the variable for its value but for its location, that is, for its ability to store a value. The right hand argument of the assignment operator is evaluated for a value and that value is placed in the location given by evaluating the left hand argument.

In this example, suppose the current value of variable i is 7. The right hand side evaluates to 10, the value 10 is therefore stored in variable i, also the value 10 is the resulting value of the overall expression.

It should come as no surprise that for values of integer type there are addition, subtraction and multiplication operators. There is also a division operator but since the operator must evaluate to an integer, it might be surprising that, for instance $2/3$ evaluates to zero! Two-thirds is not an integer, and the operator is constrained to give integer values as a result of the evaluation.

A very important class of operators are those that work on values of boolean type, and those that evaluate to boolean type even if their arguments are not boolean. For instance, the greater-than operator in the expression,

```
i > 0
```

```
{
   int  i ;
   i = 3 ;
   {
        int j ;
        j = 2 * i  ;
    }
    i = 2 * i  ;
}
```

Figure 3: Block structured, compound statemets

evaluates true if the integer contains a value larger than zero, and false otherwise. The arguments are integer type, the expression results in a value of boolean types.

2.3 Exercises

1. Auto-increment
2. $+=$ and friends
3. Logical connectives and short-circuited evaluation
4. modulus and its definition, truncating division.

2.4 Statements and block structure

A program is a sequence of *statements.* Some statements are addressed to the compiler. they are about the program as text, they do not correspond to some specific action taken when the program is run. A declaration statement is a statement of this type. Other statements are at the other extreme — they are fairly direct instructions for actions to take when the program is run. Statements made from expressions are examples of this type of statement.

Java is a *block structured language.* Several statements can be combined into a single *compound statement.* In many ways, the compound statement is treated as a single statement. This allows a complex action to be inserted anywhere that a simple action is allowed. Figure 3 gives an example of compound statement comprised of declaration and expression statements.

A *declaration statement* introduces a variable name to the compiler. The region of program text where the variable is known is the variables *scope.* The declaration statement begins the scope. The end of the compound statement directly enclosing the declaration of a variable ends the variable's scope. In figure 3, the variable i is known throughout the code fragment. The variable j is known only within the innermost compound statement. The scope of j does not include the region of code where the assignment of 3 to i is made.

```
class IfElse  {

   public static void main( String [ ] args )    {
      int i ;
      i = -1 ;
      if ( i > 0 )    {
          System.out.println("i is greater than 0") ;
      }
      else  {
           System.out.println("i is less than or equal to 0") ;
      }
   }
}
```

Figure 4: Example of branching: IfElse.java

The *expression statement* is formed by taking an expression and adding a semicolon to the end. The evaluation of the expression then forms a distinct step in the program. We have so far written simple programs which are sequences of expression statements, sometimes also declaration statements to introduce the name and type of variables to be used in expressions. These statements are evaluated one at a time, values are calculated, printed and stored in variables.

2.5 Branching

Normally, those statements that cause action do so in the sequence the statements appear as text. Branching and looping provide for alternative code flow depending on the state of the world. For the computer, the state of the world boils down to the resulting value of an expression of type boolean. The expression yields either true or false. Branching allows the programmer to attach one set of actions to a true result and another to a false result.

Branching is achieved by using the *if statement*. The statement is introduced by the keyword *if*, followed by an expression enclosed in parenthesis. The parenthesis pair is part of the syntax of the if statement, they cannot be omitted. The expression enclosed by the parenthesis must evaluate to type boolean. Following the parenthesis follows either a single statement or a compound statement. We demonstrate this second form: it is more versatile and it is better for beginners to keep syntax to a minimum. The statement or all statements in the compound statement are run if (and only if) the expression evaluates to true.

The if statement can, optionally, be followed by an *else statement*. This begins with the keyword *else* and followed by either a single statement or a compound statement. Again, we will demonstrate only the use of a compound statement. If the expression evaluates false then the else statement or compound statement is evaluated. The else statement must follow the if statement immediately, no other statements can intervene.

Consider the if statement or the combined if-else statement as a single step in the code. After

```
class Boom  {

   static final int START_OF_COUNT = 10 ;

   public static  void main( String [] args )  {
      int i ;
      i  = START_OF_COUNT ;
      while ( i>0 )  {
         System.out.println(i) ;
         i = i - 1 ;
      }
      System.out.println("Boom!") ;
   }

}
```

Figure 5: Example of looping: Boom.java

it is executed the computer moves on to the next statement, regardless whether the expression evaluated true or false.

2.6 Looping

Branches in our code mean that the program will not always do the same time each time it is run. It will modify its behavior according to circumstances. This is a very powerful mechanism. But even more powerful is the loop. Branching does not escape an obvious restriction on our code: that it will run only so long, advancing line by line from top to bottom until the last statement is run. What finally breaks this restriction is any of the looping constructs — code flow can return upwards in the text, to repeat of block of code an arbitrary number of times until a given state of the world attains. There are several constructs for looping, we discuss the *while statement*.

The while statement is introduced by the keyword *while* and is followed by an expression enclosed in parenthesis. The pair of parenthesis are part of the syntax of the while statement, they cannot be omitted. The expression enclosed by the parenthesis must evaluate to type boolean. Following the expression is a statement or a compound statement, called the *body* of the while loop. For simplicity we will consider only the case of the compound statement.

When the while statement is encountered in the flow of code, the expression is evaluated. If it evaluates to true the compound statement is executed. Afterwards the expression is again evaluated. As long as it evaluates true the body of the while loop is executed. When the expression evaluates false the body is not executed and control passes to the following statement. It is possible that the body is never executed: if on first encountering the while statement the expression yields false the body is not run.

The class Boom is defined. It includes the method main. This method is a sequence of statements,

```
public static void lineOfStars( int numberOfStars ) // method signature
// method body follows
{
     while ( numberOfStars > 0 )
     {
         System.out.print('*') ;
          numberOfStars = numberOfStars - 1 ;
     }
    System.out.println() ;
}
```

Figure 6: Example of a method: lineOfStars

run in order written, except that once the while construct is encountered, the statements it encloses will be run repeatedly until the expression $i > 0$ becomes false.

There are several other looping constructs. The most popular is the *for statement*. They can be replaced with while loops, the while loop is universal. In fact, even branching can be replaced with the while loop: set the if expression as the while expression conjuncted with a flag to prevent more than a single repetition. The interested reader should follow up this suggestion as an exercise.

2.7 Exercises

1. Switch statement
2. For loop, how they are re-written by the compiler to while loops
3. Break and continue constructs
4. Do until construct, not in java
5. Using while to do if-else statements

2.8 Method invocation

The word *method* is particular to object oriented programing. A method is not conceptually different from a function or subroutine of non-object oriented programming. But people felt it best to have a new word in order to emphasize a new outlook.

A method is comprised of a *body*, which is a compound statement, introduced by the *method signature*, which states the name of the method, the number arguments the method expects, the type of each argument, and the type of the value resulting when the method body is evaluated. There are additional keywords in the method signature, such as *static* and *public*. These have to do with the object-oriented aspect of Java and will be explained when we turn our attention to that concept.

```
class Stars {

      static final int SIZE_OF_TRIANGLE = 5 ;

      // lineOfStars method
    public static void lineOfStars( int numberOfStars ) // method signature
    // method body follows
    {
         while ( numberOfStars > 0 ) {
            System.out.print('*') ;
            numberOfStars = numberOfStars - 1 ;
          }
        System.out.println() ;
    }

   public static void main( String [ ] args ) {
        int i ;
       i = SIZE_OF_TRIANGLE ;
       while ( i > 0 )  {
            lineOfStars( i ) ;
            i = i - 1 ;
        }
   }
}
```

Figure 7: Example of method use: Star.java

The arguments listed in the the method signature are called *formals*. They are variables of the stated type whose scope is exactly the method body. The initial values of the formals are determined when the method is *invoked*, that is, when for some reason it is necessary to evaluate the method body. Typically is occurs when the method name is encountered in an expression. When a method is invoked, in the location of each formal will be an expression. The expression is evaluated and the resulting value is the initial value for the corresponding formal.

This exchange of information into a method body, that is, by evaluation of an expression in the environment of the method invoker, the copying of the resulting value into a variable whose scope is that of the method invoked, is termed *call by value*. Java is among several languages that use call by value.

The method can also return a value as a result of its evaluation. A special *return statement* inside the method directs what value to return. The signature of the method must specify the type of value which a method will return. In the case of a method is never to return a value, it is marked as returning *void type* in the signature.

Then a method name is encountered, the body is evaluated both for side-effects, for instance, things that are printed to the screen, and possibly for a value. A method which evaluates the trigonometric function sine will, when encountered in an expression, be replaced with the value of the sine of its argument.

Methods serve to set aside, one time for all, a procedure for accomplishing some desirable aim, such as computing the sine function, or, in our example, printing out a line of stars. So that they be most useful, methods have formals so that their computations can commence with different values according to the wishes of the caller. In the example Stars.java we print out a series of lines of stars of decreasing lengths. We call the same method with a different value of *numberOfStars*.

2.9 Coding and structuring with methods

Besides a technical device for the control of program flow, methods have several advantageous properties.

1. Methods serve to set aside, one time for all, a procedure for accomplishing some desirable aim, such as computing the sine function, or, in our example, printing out a line of stars. The code reads much easier and is easier to design, since the method as a simple, clear goal. It also helps the coder think of a larger task in terms of simpler tasks, each with a clear and sensible description, such as "to print a line of n stars". A method might invoke other methods to achieve that goal. The invoker of the method only has to supply the correct parameters.
2. In practice, having a method simplifies code repair. If a procedure turns out to have a defect, fixing the method responsible for the procedure fixes the behavior of all users of the method.
3. In practice, simplies code debug, serving as choke-points for code flow, and checking the argument values on entry to a method call often provides the key data required to isolate a programming error.
4. It helps control the interaction between different parts of the the code. In particular, distinct methods should interfere with each other as little as possible. The inner workings of a method should be that method's private business. Other methods should have no dependence of on these inner workings, not expose its inner workings to other methods. In general, how the method varies in its action from use to use should depend only on the value of the formals, not on a vast variety of hidden and widely scattered externalities.

2.10 Recursion

A method my invoke other methods, including itself. This last is called *recursion*, and in a theoretical sense, gives method invocation its real power. After all, despite the "psychological" virtues enumerated above, a method invocation can be replaced with the method body, and its invocations with those bodies, and so forth, until we have a program consisting only of loops, branches and straight-line code. If fact, if none of the methods used loops, we would have a simple straight-line program, limited in its power as are all straight-line programs.

However a recursion cannot be unrolled, since every substitution of the body of a method for the method name reintroduces the name. Recursion can also occur indirectly. Say method *one* invokes

```
class SumOfOddNumbers {
       static final int NUMBER_OF_ODDS = 5 ;

       static int sumOdds( int anOddNumber ) {
            if ( anOddNumber <= 1 ) {
                return 1 ; }
           else {
            return anOddNumber + sumOdds( anOddNumber - 2 ) ;
          }
      }

     public static void main( String [ ] args ) {
           System.out.println( sumOdds(2 * NUMBER_OF_ODDS -1) ) ;
    }
}
```

Figure 8: Example of recursion: SumOfOddNumbers.java

method *two* which invokes method *one*. Therefore, if we are to understand method invocation as something other than a psychological convenient packaging of actions we must understand recursion. Also, the precise apparatus creating the formals and other local variables in a method are most clearly demonstrated when considering recursion. Therefore, for a proper understanding of the apparatus of method invocation, recursion must be considered.

The program SumOfOddNumbers.java demonstrates recursion. The method main calculates the n-th odd number, according to the static final variable `NUMBER_OF_ODDS`. The sumOdds method is recursive. Its job is to add the odd numbers from 1 up to the value in the formal variable `anOddNumber`. If `anOddNumber` is 1, then the solution is obvious, and the procedure returns the answer, the value 1. In any other case, sumOdds first figures out what the sum of the odd numbers from 1 up to `anOddNumber` − 2 and then adds to that the value of `anOddNumber`.

What is important to understand is that each time `sumOdds` is called, their is a new and separate variable `anOddNumber` created. Although the code seems to jump back and repeat itself, a new data environment is created for each invocation of a method (recursive or not). This environment is called a *stack frame*. It contains all the variables local to the method as well as the formals. As methods call methods, the frames stack up like plates, one plate covering another. Only the variables of the top most frame are visible; and even if two frames contain variables of the same name they are not confused — only variables of the topmost frame are visible, only those variables are used.

Compare this to the notion of scope. `anOddNumber` is in scope throughout the method. That means the name is known. It refers to a variable. Which variable depends on the currently active stack frame. Hence scope refers to the knowledge of a name, not of a variable. For certain, behind the name lies a variable, but which variable will be determined when the program is run, not when it is compiled.

On return from a method, concurrent with the return of control back to the point in the invoker at which the method was invoked, the stack frame created by the invocation is removed, uncovering the older stack frame, its variables and their values recovered as from the moment it was covered. Control is returned and so is the data environment. In the case of recursion, each frame resembles the other, the same named local variables and the same named formals —- but they are different.

The variable is created when the method is invoked and destroyed when the method is completed. The time during which the variable exists is called the *lifetime* of the variable. This completes the description of the four important properties of a variable: name, type, scope and lifetime. The lifetime of a variable is entirely a concept associated with the running of a program. Scope on the other hand, is associated with the compiling of the program —- it is entirely determined by the program text.

3 Meaning

3.1 Assertions, preconditions and postconditions

An *assertion* is a statement which is true about the world at the moment it is asserted. It is placed in the code to ascertain correctness. It is never correct for an assertion to fail, to evaluate false. Sometimes assertions are written in prose, as comments. These have no real effect on the running of the program. Different languages have different facilities for writing assertions into the program code and for halting program execution if the assertion fails. A common implementation allows for all assertions to be ignored or evaluated according to how the program is built. In this way, when the programmer is confident that all assertions will yield true, the program is built so that they are not evaluated. However, if a problem arises, or if the program is in a test phase, the program is built with the assertions evaluated. In this way the first moment that something goes wrong, an assertion fails, the program execution halts and a message indicates which assertion failed.

Assertions are used in various ways. Two very typical usages are as *pre-conditions* and as *post-conditions*. A pre-condition is an assertion before entering a block of code which summarizes what must be true in the state of the world in order that the block of code execute successfully. These are the assumptions that the programmer has made when writing the code inside the block. The code is not incorrect if it fails when run in an environment where the preconditions are not satisfied. That would be the fault of the code preceding the code block. They did not achieve the pre-conditions. However, if the pre-conditions are achieved and the code block fails, then something is wrong with the code block, or perhaps the programmer might want to add another pre-condition, tighten the rules of entry, so to speak, to the block of code.

A post-condition is an assertion after leaving a block of code. It summarizes what has been accomplished by the code. Typically it is an assertion which would not necessarily evaluate to true if the code block were not run; running the code block guarantees the condition.

Code is written with interlocking assertions. The pre-conditions of one block of code are the post-conditions of the directly preceding block; and the post-conditions of another block match the pre-conditions of the following block. The pieces of code in between are transformations, calculations, which starting from the assumptions of the pre-condition manipulate the world to satisfy the post-conditions.

A further illustration is integration of pre and post-conditions with methods. The pre-conditions for a method are the requirements on the arguments that the method work as planned. The post-condition is what you can expect the method to accomplish on return.

3.2 Loop invariants

The most powerful demonstration of the use of assertions for the writing of correct code is the discipline of *loop invariants*. This consists of an assertion which captures at once the precondition for entering each iteration of the loop body and the postcondition on leaving each iteration of the loop body. It is expected that at some point during the loop body, in order to advance towards the ultimate goal of the loop, variable values are in the process of being adjusted and at that moment the loop invariant would not be true. It cannot be asserted at that time. However, before the bottom of the loop body is encountered all will be set right, and the loop invariant can be asserted.

This simplifies the analysis of loop correctness immeasurably. The programmer, in considering the correctness of the loop, focuses entirely on a single pass through the loop, and whether, assuming the loop invariant is true at the start of the iteration, it is again true at the end of the iteration. The iterations then glue themselves together without complicated reasoning which crosses over loop iterations.

It is important to notice, and part of the loop invariant discipline, that the loop invariant does not include any concern for the termination of the loop. Only that it be correct for so long as it runs. Termination of a loop is a separate issue. In fact, termination of a loop is a very difficult issue and deserves to be taken aside and thought about cleanly and in isolation. The loop invariant says that if the loop terminates then it will terminate with the loop invariant true. The invariant and terminate should coordinate so that if the loop terminates, the fact the invariant is true implies that the loop has, overall, achieved its goal. I write this in a pseudo-mathematically formula,

$$\text{loop invariant} \; + \; \text{termination} \; = \; \text{goal}$$

Of loops can be created so that termination is obvious. For instance, a variable is decremented each iteration through the loop and the termination condition is the variable value falling below a certain number. This is a good way to set up a loop. Other times termination is not obvious, and the argument for termination is mathematically sophisticated. This is to be avoided if possible, but for some problems it cannot be avoided. In these situations it is most helpful that reasoning about termination is cleanly separated from the loop invariant, so that each matter can be taken up in turn.

The example shows the methodology of loop invariants to the calculation of the greatest common divisor. The greatest common divisor of two positive integers a and b is the largest number c which divides evenly both a and b. One meets this concept first when reducing fractions. For instance the greatest common divisor of 42 and 35 is 7, so 35/42 is written in lowest form as 5/6 after dividing both numerator and denominator by the gcd (greatest common divisor).

In symbols, the gcd of a and b is written as a function gcd(a,b). The idea of the algorithm is that $gcd(a, b) = gcd(b, a\%b)$, where $a\%b$ is the remainder after a is divided by b. Note $a = gcd(a, 0)$, and by repeating the above reduction we will eventually get to this easy case.

3.3 Transformation to equivalent code

short circuited logicals; equivalence of code; boolean identities

4 Classes and types

```
class GreatestCommonDivisor {

    static final int A = 42 ;
    static final int B = 35 ;

    public static void main( String [] args ) {
       // precondition, A >= B >= 0.
       int a ;
       int b ;
       int gcd ;

       a = A ;
       b = B ;

       // Loop Invariant: (1) gcd(a,b) = gcd(A,B)
       //             and (2) a>=b>=0.

       // ASSERT Loop Invariant (why is it true?)
       while ( b > 0 ) {
         int t ; // just a temporary (Question: what is the scope of t?)
         t = a%b ;
         a = b ;
         b = t ;
         // ASSERT Loop Invariant, (1) gcd(a,b) = gcd(A,B)
         // and (2) a >= b >= 0
       }

       // Loop Invariant + Termination = Goal
       gcd = a ; // because gcd(a,0) is a

       // postcondition gcd = GreatestCommonDivisor of A and B
       System.out.println("The GCD of "+A+" and "+B+" is "+gcd) ;
    }
}
```

Figure 9: Example: Loop invariant