

2

Conditional Structures and Loops

2.1 Instruction workflow

In this chapter, we start by describing how programmers can control the execution paths of programs using various branching conditionals and looping structures. These branching/looping statements act on blocks of instructions that are sequential sets of instructions. We first explain the single choice **if else** and multiple choice **switch case** branching statements, and then describe the structural syntaxes for repeating sequences of instructions using either the **for**, **while** or **do while** loop statements. We illustrate these key conceptual programming notions using numerous sample programs that emphasize the program workflow outcomes. Throughout this chapter, the leitmotiv is that the execution of a program yields a workflow of instructions. That is, for short:

Program runtime = Instruction workflow
--

2.2 Conditional structures: Simple and multiple choices

2.2.1 Branching conditions: `if ... else ...`

Programs are not always as simple as plain sequences of instructions that are executed step by step, one at a time on the processor. In other words, programs are not usually mere monolithic blocks of instructions, but rather compact structured sets of instruction blocks whose executions are decided on the fly. This gives a program a rich set of different instruction workflows depending on initial conditions.

Programmers often need to check the status of a computed intermediate result to branch the program to such or such another block of instructions to pursue the computation. The elementary branching condition structure in many imperative languages, including Java, is the following `if else` instruction statement:

```
if (booleanExpression)
{BlockA}
else
{BlockB}
```

The boolean expression `booleanExpression` in the `if` structure is first *evaluated* to either `true` or `false`. If the outcome is `true` then `BlockA` is executed, otherwise it is `BlockB` that is selected (boolean expression evaluated to `false`). Blocks of instructions are delimited using braces `{...}`. Although the curly brackets are *optional* if there is only a single instruction, we recommend you set them in order to improve code readability. Thus, using a simple `if else` statement, we observe that the same program can have different execution paths depending on its initial conditions. For example, consider the following code that takes two given dates to compare their order. We use the branching condition to display the appropriate console message as follows:

```
int h1 =..., m1 =..., s1 =...; // initial conditions
int h2 =..., m2 =..., s2 =...; // initial conditions
int hs1 = 3600*h1 + 60*m1 + s1;
int hs2 = 3600*h2 + 60*m2 + s2;
int d=hs2-hs1;

if (d>0) System.out.println("larger");
    else
        System.out.println("smaller or identical");
```

Note that there is no `then` keyword in the syntax of Java. Furthermore, the `else` part in the conditional statement is optional:

```
if (booleanExpression)
{BlockA}
```

Conditional structures allow one to perform various status checks on variables to branch to the appropriate subsequent block of instructions. Let us revisit the quadratic equation solver:

Program 2.1 Quadratic equation solver with user input

```
import java.util.*;
class QuadraticEquationRevisited
{
    public static void main(String [] arg)
    {
        Scanner keyboard=new Scanner(System.in);

        System.out.print("Enter a,b,c of equation ax^2+bx+c=0:");
        double a=keyboard.nextDouble();
        double b=keyboard.nextDouble();
        double c=keyboard.nextDouble();

        double delta=b*b-4.0*a*c;
        double root1, root2;

        if (delta >= 0)
        {
            root1= (-b-Math.sqrt(delta))/(2.0*a);
            root2= (-b+Math.sqrt(delta))/(2.0*a);
            System.out.println("Two real roots:"+root1+" "+root2);
        }
        else
        {System.out.println("No real roots");}
    }
}
```

In this example, we asserted that the computations of the roots `root1` and `root2` are possible using the fact that the discriminant `delta >= 0` in the block of instructions executed when expression `delta >= 0` is `true`. Running this program twice with respective user keyboard input 1 2 3 and -1 2 3 yields the following session:

```
Enter a,b,c of equation ax^2+bx+c=0:1 2 3
No real roots
Enter a,b,c of equation ax^2+bx+c=0:-1 2 3
Two real roots:3.0 -1.0
```

In the `if else` conditionals, the boolean expressions used to select the appropriate branchings are also called *boolean predicates*.

2.2.2 Ternary operator for branching instructions:

Predicate ? A : B

In Java, there is also a special compact form of the `if else` conditional used for variable assignments called a ternary operator. This ternary operator `Predicate ? A : B` provided for branching assignments is illustrated in the sample code below:

```
double x1=Math.PI; // constants defined in the Math class
double x2=Math.E;
double min=(x1>x2)? x2 : x1; // min value
double diff= (x1>x2)? x1-x2 : x2-x1; // absolute value
System.out.println(min+" difference with max="+diff);
```

Executing this code, we get:

```
2.718281828459045 difference with max=0.423310825130748
```

The compact instruction

```
double min=(x1>x2)? x2 : x1;
```

...is therefore equivalent to:

```
double min;
if (x1>x2) min=x2;
else min=x1;
```

Figure 2.1 depicts the schema for unary, binary and ternary operators.

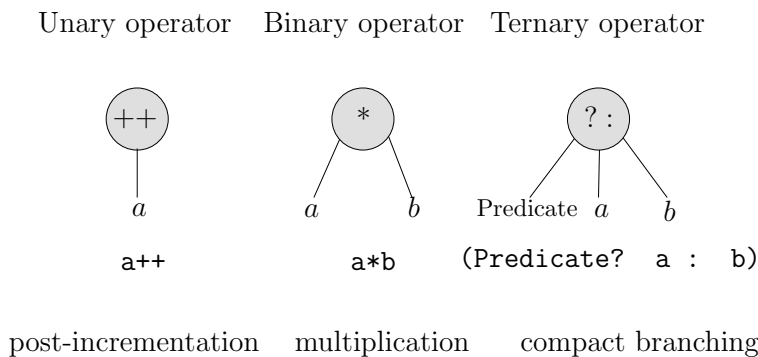


Figure 2.1 Visualizing unary, binary and ternary operators

2.2.3 Nested conditionals

Conditional structures may also be nested yielding various complex program workflows. For example, we may further enhance the output message of our former date comparison as follows:

```
int h1 =..., m1 =..., s1 =...;
int h2 =..., m2 =..., s2 =...;
int hs1 = 3600*h1 + 60*m1 + s1;
int hs2 = 3600*h2 + 60*m2 + s2;
int d=hs2-hs1;

if (d>0) {System.out.println("larger");}
else
    {if (d<0)
      {System.out.println("smaller");}
      else
      {System.out.println("identical");}
    }
```

Since these branching statements are all single instruction blocks, we can also choose to remove the braces as follows:

```
if (d>0) System.out.println("larger");
else
if (d<0)
    System.out.println("smaller");
else
    System.out.println("identical");
```

However, we do not recommend it as it is a main source of errors to novice programmers. Note that in Java there is *no* shortcut¹ for **else if**. In Java, we need to write plainly **else if**. There can be any arbitrary *level* of nested **if else** conditional statements, as shown in the generic form below:

```
if (predicate1)
    {Block1}
else
    {
        if (predicate2)
            {Block2}
        else
        {
            if (predicate3)
                {Block3}
            else
            {
                ...
            }
        }
    }
```

¹ In some languages such as Maple®, there exists a dedicated keyword like **elif**.

```
}
```

In general, we advise to always take care with boolean predicates that use the equality tests `==` since there can be numerical round-off errors. Indeed, remember that machine computations on reals are done using single or double precision, and thus the result may be truncated to fit the formatting of numbers. Consider for example the following tiny example that illustrates numerical imprecisions of programs:

```
class RoundOff
{
    public static void main(String[] arg)
    {
        double a=1.0d;
        double b=3.14d;
        double c=a+b;

        if (c==4.14) // Equality tests are dangerous!
        {
            System.out.println("Correct");
        }
        else
        {
            System.out.println("Incorrect. I branched on the wrong
                block!!!");
            System.out.println("a="+a+" b="+b+" a+b=c="+c);
            // unexpected behavior may follow...
        }
    }
}
```

Running this program, we get the surprising result caused by numerical precision problems:

```
Incorrect. I branched on the wrong block!!!
a=1.0 b=3.14 a+b=c=4.140000000000001
```

This clearly demonstrates that equality tests `==` in predicates may be harmful.

2.2.4 Relational and logical operators for comparisons

The *relational operators* (also called comparison operators) that evaluate to either `true` or `false` are the following ones:

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

One of the most frequent programming errors is to use the equality symbol = instead of the relational operator == to test for equalities:

```
int x=0;

if (x=1) System.out.println("x equals 1");
else System.out.println("x is different from 1");
```

Fortunately, the compiler generates an error message since in that case types `boolean/int` are incompatible when performing type checking in the expression `x=1`. But beware that this will not be detected by the compiler in the case of boolean variables:

```
boolean x=false;
if (x=true) System.out.println("x is true");
else System.out.println("x is false");
```

In that case the predicate `x=true` contains an assignment `x=true` and the evaluated value of the “expression” yields `true`, so that the program branches on the instruction `System.out.println("x is true");`.

A boolean predicate may consist of several relation operators connected using *logical operators* from the table:

&&	and
	or
!	not

The logic truth tables of these connectors are given as:

&&	true	false
true	true	false
false	false	false

	true	false
true	true	true
false	true	false

Whenever logical operators are used in predicates, the boolean expressions are evaluated in a lazy fashion as follows:

- For the `&&` connector, in `Expr1 && Expr2` do not evaluate `Expr2` if `Expr1` is evaluated to `false` since we already know that in that case that `Expr1 && Expr2 = false` whatever the `true/false` outcome value of expression `Expr2`.

- Similarly, for the `||` connector, in `Expr1 || Expr2` do not evaluate `Expr2` if `Expr1` is evaluated to `true` since we already know that in that case `Expr1 || Expr2 = true` whatever the `true/false` value of expression `Expr2`.

The lazy evaluation of boolean predicates will become helpful when manipulating arrays later on. For now, let us illustrate these notions with a simple example:

Program 2.2 Lazy evaluation of boolean predicates

```
class LazyEvaluation{
    public static void main (String [] args)
    {
        double x=3.14, y=0.0;
        boolean test1, test2;

        // Here division by zero yields a problem
        // But this is prevented in the && by first checking
        // whether the denominator is
        // zero or not
        if ((y!=0.0) && (x/y>2.0))
            { // Do nothing
              ; }
            else
            { // Block
              test1=(y!=0.0);
              test2=(x/y>2.0);

              System.out.println("Test1:"+test1+" Test2:"+test2);
              System.out.println("We did not evaluate x/y that is
                equal to "+(x/y));
            }

        // Here, again we do not compute x/y since the first term
        // is true
        if ((y==0.0) || (x/y>2.0))
            { // Block
              System.out.println("Actually, again, we did not
                evaluate x/y that is equal to "+(x/y));
            }
    }
}
```

Running this program, we get the following console output:

```
Test1:false Test2:true
We did not evaluate x/y that is equal to Infinity
Actually, again, we did not evaluate x/y that is equal to Infinity
```


2.2.5 Multiple choices: switch case

The nested **if else** conditional instructions presented in § 2.2.3 are somehow difficult to use in case one would like to check that a given variable is equal to such or such a value. Indeed, nested blocks of instructions are difficult to properly visualize on the screen. In the case of **multiple choices**, it is better to use the **switch case** structure that branches on the appropriate set of instructions depending on the value of a given expression. For example, consider the code:

```
class ProgSwitch
{
    public static void main(String arg []) {
        System.out.print("Input a digit in [0..9]:");
        Scanner keyboard=new Scanner(System.in);
        int n=keyboard.nextInt();
        switch(n)
        {
            case 0: System.out.println("zero"); break;
            case 1: System.out.println("one"); break;
            case 2: System.out.println("two"); break;
            case 3: System.out.println("three"); break;
            default: System.out.println("Above three!");
            break;
        }
    }
}
```

The conditional statement **switch** consider the elementary expression n of type **int** and compare it successively with the first case: **case 0**. This means that **if (n==0)Block1 else \{ ... \}**. The set of instructions in a **case** should end with the keyword **break**. Note that there is also the **default** case that contains the set of instructions to execute when none of the former cases were met. The formal syntax of the multiple choice **switch case** is thus:

```
switch( TypedExpression )
{
    case C1:
        SetOfInstructions1;
        break;
    case C2:
        SetOfInstructions2;
        break;
    ...
    case Cn:
        SetOfInstructionsn;
        break;
    default:
        SetOfDefaultInstructions;
}
}
```

Multiple choice `switch` conditionals are often used by programmers for displaying messages²:

Program 2.3 Demonstration of the `switch case` statement

```
int dd=3; // 0 for Monday, 6 for Sunday

switch(dd)
{
case 0:
    System.out.println("Monday"); break;
case 1:
    System.out.println("Tuesday"); break;
case 2:
    System.out.println("Wednesday"); break;
case 3:
    System.out.println("Thursday"); break;
case 4:
    System.out.println("Friday"); break;
case 5:
    System.out.println("Saturday"); break;
case 6:
    System.out.println("Sunday"); break;
default:
    System.out.println("Out of scope!");
}
```

2.3 Blocks and scopes of variables

2.3.1 Blocks of instructions

A block of instructions is a set of instructions that is executed sequentially. Blocks of instructions are delimited by braces, as shown below:

```
{
// This is a block
// (There are no control structures inside it)
Instruction1;
Instruction2;
...
}
```

A block is semantically interpreted as an atomic instruction at a macroscopic level when parsing.

² Or used for translating one type to another when used in functions

2.3.2 Nested blocks and variable scopes

Blocks can be nested. This naturally occurs in the case of **if-else** structures that may internally contain other conditional structures. But this may also be possible without conditional structures for controlling the scope of variables. Indeed, variables defined in a block are defined for all its sub-blocks. Thus a variable cannot be redefined in a sub-block. Moreover variables defined in sub-blocks cannot be accessed by parent blocks as illustrated by the following example:

```
class NestedBlock
{
    public static void main(String [] arg)
    {
        int i=3;
        int j=4;

        System.out.println("i="+i+" j="+j);

        {
            // Cannot redefine a variable i here
            int ii=5;
            j++;
            i--;
        }

        System.out.println("i="+i+" j="+j);
        // Cannot access variable ii here
    }
}
```

```
i=3 j=4
i=2 j=5
```

Finally note that single instructions in control structures such as **if-else** are interpreted as implicit blocks where braces are omitted for code readability.

2.4 Looping structures

Loop statements are fundamental structures for iterating a given sequence of instructions, repeating a block of instructions. Java provides three kinds of constructions for ease of programming, namely: **while**, **for** and **do-while**. Theoretically speaking, these three different constructions can all be emulated with a **while** statement. We describe the semantic of each structure by illustrating it with concrete examples.

2.4.1 Loop statement: while

The syntax of a **while** loop statement is as follows:

```
while (boolean_expression)
{block_instruction;}
```

This means that while the **boolean_expression** is evaluated to **true**, the sequence of instructions contained in the **block_instruction** is executed.

Consider calculating the *greatest common divisor* (gcd for short) of two integers a and b . That is, the largest common divisor c such that both a and b can be divided by c . For example, the GCD of $a = 30$ and $b = 105$ is 15 since $a = 2 \times 3 \times 5$ and $b = 5 \times 3 \times 7$. Euclid came up with a simple algorithm published for solving this task. The algorithm was reported in his books *Elements* around³ 300 BC. Computing the GCD is an essential number operation that requires quite a large amount of computation for large numbers. The GCD problem is related to many hot topics of cryptographic systems nowadays. Euclid's algorithm is quite simple: If $a = b$ then clearly the GCD of a and b is $c = a = b$. Otherwise, consider the largest integer, say a without loss of generality, and observe the important property that

$$\text{GCD}(a, b) = \text{GCD}(a - b, b).$$

Therefore, applying this equality reduces the total sum $a + b$, and at some point, after k iterations, we will necessarily have $a_k = b_k$. Let us prove a stronger result: $\text{GCD}(a, b) = \text{GCD}(b, a \bmod b)$.

Proof

To see this, let us write $a = qb + r$ where q is the quotient of the Euclidean division and r its remainder. Any common divisor c of a and b is also a common divisor of r : Indeed, suppose we have $a = ca'$ and $b = cb'$, then $r = a - qb = (a' - qb')c$. Since all these numbers are integers, this implies that r is divisible by c . It follows that the greatest common divisor g of a and b is also the greatest common divisor of b and r . $\square$

Let us implement this routine using the **while** loop statement. The terminating state is when $a = b$. Therefore, while $a \neq b$ (written in Java as **a!=b**), we retrieve that smaller number to the larger number using a **if** conditional structure. This gives the following source code:

³ It is alleged that the algorithm was likely already known in 500 BC.

Program 2.4 Euclid's Greatest Common Divisor (GCD)

```

class GCD {
public static void main(String [] arg)
{
int a;
int b;
while (a!=b)
{
if (a>b) a=a-b;
else b=b-a;
}
System.out.println(a); // or b since a=b
}
}

```

Running this program for $a = 30$ and $b = 105$ yields $\text{GCD}(a, b) = 15$.

Euclid's algorithm has a nice *geometric* interpretation: Consider an initial rectangle of width a and height b . Bisect this rectangle as follows: Choose the smallest side, and remove a square of that side from the current rectangle. Repeat this process until we get a square: The side length of that square is the GCD of the initial numbers. Figure 2.2 depicts this “squaring” process.

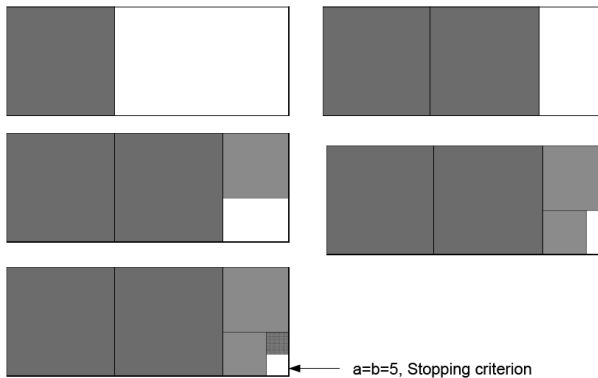


Figure 2.2 A geometric interpretation of Euclid's algorithm. Here, illustrated for $a = 65$ and $b = 25$ ($\text{GCD}(a, b) = 5$)

2.4.2 Loop statement: do-while

Java provides another slightly different syntax for making iterations: the **do** loop statement. The difference with a **while** statement is that we execute at least once the sequence of instructions in a **do** statement, whereas this might not be the case of a **while** statement, depending on the evaluation of the boolean predicate. The syntax of a **do** structure is as follows:

```
do
{block_instruction;}
while (boolean_expression);
```

That is, the `boolean_expression` is evaluated after the block of instructions, and not prior to its execution, as it is the case for `while` structures. Consider computing the square root $\sqrt{a}$ of a non-negative number a using Newton's method. Newton's method finds the closest root to a given initial condition x_0 of a smooth function by iterating the following process: Evaluate the tangent equation of the function at x_0 , and intersect this tangent line with the x -axis: This gives the new value x_1 . Repeat this process until the difference between two successive steps go beyond a prescribed threshold (or alternatively, repeat k times this process). For a given value x_n , we thus find the next value x_{n+1} by setting the y -ordinate of the tangent line at $(x_n, f(x_n))$ to 0:

$$y = f'(x_n)(x - x_n) + f(x_n) = 0.$$

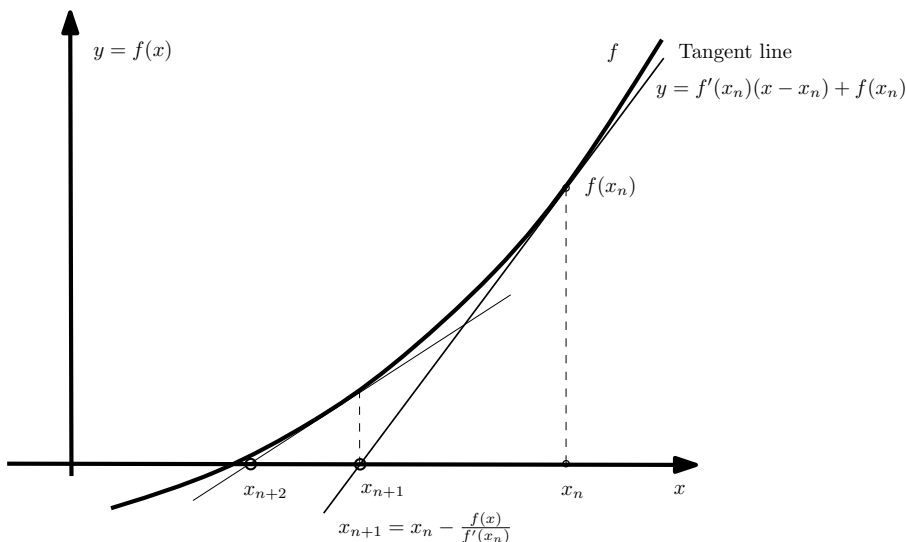


Figure 2.3 Newton's method for finding the root of a function

It follows that we get:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Figure 2.3 illustrates this root finding process. Let us use Newton's method to calculate the square root function of a by finding the root of equation $f(x) = x^2 - a$. We implement the loop using a `do` structure as follows:

Program 2.5 Newton's approximation algorithm

```
double a = 2.0, x, xold;
x = a;
do{
  xold = x;
  // compute one iteration
  x = (xold+a/xold)/2.0;
  System.out.println(x);
} while(Math.abs(x-xold) > 1e-10);
```

```
1.5
1.4166666666666665
1.4142156862745097
1.4142135623746899
1.4142135623730949
```

Newton's method is provably converging very fast under mild assumptions.

2.4.3 Loop statement: for

Often programmers need to repeat a sequence of instructions by changing some variables by a given increment step. Although this can be done using the former **while/do** structures, Java provides a more convenient structure: the **for** loop. The generic syntax of a **for** structure is as follows:

```
for(initialCondition; booleanPredicate; update)
{
  block_instructions;
}
```

For example, consider computing the cumulative sum S_n of the first n integers:

$$S_n = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}.$$

We have the recurrence equation: $S_n = n - 1 + S_{n-1}$ with $S_0 = 0$. Therefore to compute this cumulative sum, we start in reverse from S_0 and get S_i by adding $i - 1$ to S_{i-1} for all $i \in \{1, \dots, n\}$. Let us use the **for** structure as follows:

Program 2.6 Cumulative sum

```
class ForLoop
{
  public static void main(String args[])
  {
    int i, n=10;
    int cumulLoop=0;
    for (i=0; i<n; i++) {cumulLoop+=i;}
```

```

int cumul=(n*(n-1))/2; // closed-form solution
System.out.println(cumulLoop+" closed-form:"+cumul);
}
}

```

To give yet another usage of the **for** loop, consider computing an approximation of π by a Monte-Carlo simulation. We draw uniformly random points in a unit square, and count the number of points falling inside the unit disk centered inside the square. The ratio of the points falling inside this square to the overall number of points gives a good approximation of $\frac{\pi}{4}$. We draw a uniform random number in $[0, 1)$ in Java using the function **random()** of the **Math** class. This kind of Monte-Carlo simulation is extremely useful to compute complex integrals, and is often used in the finance industry. Let us give the code for approaching π :

Program 2.7 Approaching π by Monte-Carlo simulation

```

class MonteCarloPI
{
public static void main(String [] args)
{
int iter = 10000000; // # iterations
int hits = 0;
for (int i = 0; i < iter; i++)
{
double rX = 2*Math.random() - 1.0;
double rY = 2*Math.random() - 1.0;
double dist = rX*rX + rY*rY;
if (dist <= 1.0) // falls inside the disk
hits++;
}
double ratio = (double)hits/iter; // Ratio of areas
double area = ratio * 4.0;
System.out.println("Estimation of PI: " + area+ " versus
    library PI "+Math.PI);
}
}

```

Unfortunately running this code gives a poor approximation of π since we get only a few correct digits, even after drawing 10^7 points.

Estimation of PI: 3.1413544 versus library PI 3.141592653589793

2.4.4 Boolean arithmetic expressions

A category of arithmetic expressions that are especially useful for writing predicates in loop structures are boolean expressions. Although they are not

usually used in plain assignments, they also make perfect sense as illustrated by the program below:

Program 2.8 Boolean arithmetic expression

```
class Boolean{
public static void main(String [] args)
{
boolean b1 = (6-2) == 4;
boolean b2 = 22/7 == 3+1/7.0 ;
boolean b3 = 22/7 == 3+ 1/7;
System.out.println(b1); // true
System.out.println(b2); // false
System.out.println(b3); // true
}
}
```

2.5 Unfolding loops and program termination

2.5.1 Unfolding loops

When executing a program that contains loop structures, we can unroll these loops manually. Compilers actually do it sometimes to optimize the generated bytecode.

2.5.2 Never ending programs

Once programmers first experience loops, a major issue arises: Does the program terminate? It is indeed quite easy to write never ending programs by writing loops that execute forever as illustrated below:

```
int i=0;
while (true)
i++;
```

Always make sure when you write a **for** structure that the boolean expression will evaluate to **false** at some stage. Take care to avoid mistyping problems such as:

```
for(i=0;i>=0;i++)
; // common mistyping error in the boolean predicate
```

... and prefer to use curly brackets instead of the semi-colon for single-instruction blocks:

```
for(i=0;i>=0;i++)
{ }
```

2.5.3 Loop equivalence to universal while structures

As mentioned earlier, the three loop structures in Java are all equivalent to a universal **while** structure. These different loop syntaxes are provided to help programmers quickly write code.

```
for(instructionInit; booleanCondition; instructionUpdate) block_instruction;
```

```
instructionInit;
while (booleanCondition)
block_instruction3; instructionUpdate;
```

2.5.4 Breaking loops at any time with break

We can voluntarily escape loops at *any time* by using the keyword **break**. This special instruction is useful for example when we ask users to input any given number of data.

2.5.5 Loops and program termination

Consider the following sequence $\{u_i\}_i$ of integers numbers as follows:

$$u_{n+1} = \begin{cases} u_n/2 & \text{if } n \text{ is even,} \\ 3u_n + 1 & \text{otherwise.} \end{cases},$$

initialized for any given $u_0 \in \mathbb{N}$.

For example, let $u_0 = 14$. Then $u_1 = 14$, $u_2 = 7$, $u_3 = 22$, $u_4 = 11$, $u_5 = 34$, $u_6 = 17$, $u_7 = 52$, $u_8 = 26$, $u_9 = 13$, $u_{10} = 40$, $u_{11} = 20$, $u_{12} = 10$, $u_{13} = 5$, $u_{14} = 16$, $u_{15} = 8$, $u_{16} = 4$, $u_{17} = 2$, $u_{18} = 1$. Once 1 is reached the sequence cycles $1, 4, 2, 1, 4, 2 \dots$. It is conjectured but not yet proved that for any $u_0 \geq 1$ the sequence reaches in a finite step number 1. We can numerically check that this conjecture holds for a given number $u_0 = n$ using the following **do** loop structure:

Program 2.9 Syracuse's conjecture

```
do{
if ((n%2)==0)
```

```
n/=2;// divide by 2
else
n=3*n+1;
} while (n>1);
```

However one has not yet managed to successfully prove that this program will eventually stop for *any* given n . It is a hard mathematical problem that is known in the literature by the name of Syracuse's conjecture or $3x + 1$ problem.⁴

This simple toy problem raises the fundamental *halting problem* famous in theoretical computer science. Loosely speaking, Gödel proved that there is *no* program that can decide whether any given program will stop after a finite number of instructions or not. This important theoretical result, which is one of the pillars of computer science, will be further explained using a simple contradiction argument in Chapter 3.8.

2.6 Certifying programs: Syntax, compilation and numerical bugs

A program that compiles without reporting *any* error message is a *syntactically* correct program. Beware that because of the language flexibility provided by its high-level semantic, some obscure codes⁵ compile. These obscure codes are often very difficult for humans to understand. To get a flavor, consider for example the snippet:

Program 2.10 Syntactically correct program

```
int i=3;
// syntax below is valid! guess its result?
int var=i+++i;
```

This program compiles and is valid since the expression $i+++i$ is well-formed. How did the compiler interpret it? Well, first the compiler put parenthesis from the operator priority rule: $(i++)+i$. Then it first evaluated this expression by performing the post-incrementation $i++$ (so that it returns 3 for this expression but now i stores value 4). Finally, it adds to the value of i 3 so that we get $3 + 4 = 7$.

Even when a simple human-readable program compiles, it becomes complex for humans to check whether the input fits all branching conditions. In other words, are all input cases considered so that the program does not have to

⁴ See <http://arxiv.org/abs/math/0608208/> for some annotated bibliographic notes.

⁵ Hackers love them.

process “unexpected” data? This can turn out to be very difficult to assert for moderate-size programs. For example, consider the quadratic equation solver:

Program 2.11 Quadratic equation solver

```
import java.util.*;
class QuadraticEquationScanner
{
public static void main(String [] arg)
{
double a,b,c; // choose a=1, b=1, c=1
Scanner input=new Scanner(System.in); input.useLocale(Locale.
    US);
a=input.nextDouble();
b=input.nextDouble();
c=input.nextDouble();
double delta=b*b-4.0*a*c;
double root1, root2;
// BEWARE: potentially Not a Number (NaN) for neg.
    discriminant!
root1= (-b-Math.sqrt(delta))/(2.0*a);
root2= (-b+Math.sqrt(delta))/(2.0*a);
System.out.println("root1="+root1+" root2="+root2);
}
}
```

The problem with that program is that we may compute roots of negative numbers. Although mathematically this makes sense with imaginary numbers $\mathbb{C}$, this is not the case for the function `Math.sqrt()`. The function returns a special number called NaN (standing for Not a Number) so that the two roots may be equal to NaN. It is much better to avoid that case by ensuring with a condition that `delta` is greater or equal to zero:

```
if (delta >= 0.0d)
{
    root1= (-b-Math.sqrt(delta))/(2.0*a);
    root2= (-b+Math.sqrt(delta))/(2.0*a);
    System.out.println("root1="+root1+" root2="+root2);
}
else
{System.out.println("Imaginary roots!");}
```

The rule of thumb is to write easy-to-read code and adopt conventions once and for all. For example, always put a semi-colon at the end of instructions, even if it is not required (atomic blocks). Always indent the source code to better visualize nested structures with braces `{}`. Take particular care of equality test `==` with assignment equal symbol `=` (type checking helps find some anomalous situations but not all of them).

Finally, let us insist that even if we considered all possible input cases and wrote our codes keeping in mind that they must also be human-readable, it

is impossible for us to consider all numerical imprecisions that can occur.⁶ Consider the following example:

Program 2.12 A simple numerical bug

```
// Constant
final double PI = 3.14;
int a=1;
double b=a+PI;
if (b==4.14) // Equality test are dangerous!!!
    System.out.println("Correct result");
else
{System.out.println("Incorrect result");
System.out.println("a="+a+" b="+b+" PI="+PI);
}
```

This code is dangerous because, mathematically speaking, it is obvious that $a+b = 4.14$ but because of the finite representation of numbers in machine (and their various formatting), this simple addition yields an approximate result. In practice, the first lesson we learn is that we always need to very cautiously use equality tests on reals. The second lesson is that proofs of programs should be fully automated. This is a very active domain of theoretical computer science that will bring novel solutions in the 21st century.

2.7 Parsing program arguments from the command line

So far we have initialized programs either by interactively asking users to enter initial values at the console, or by plugging these initial values directly into the source code. The former approach means that we have high-latency programs since user input is “slow.” The latter means that programs lack flexibility since we need to recompile the code every time we would like to test other initial parameter conditions.

Fortunately, programs in Java can be executed with *arguments* given in the command line. These arguments are stored in the array `arg` of the `main` function:

```
public static void main (String[] args)
```

These arguments are stored as strings `args[0]`, `args[1]`, etc. Thus even if we enter numbers like “120” and “28” in the command line:

⁶ Some software packages such as Astrée used in the airplane industry do that automatically to certify code robustness. See <http://www.astree.ens.fr/>

```
prompt gcd 120 28
```

These numbers are in fact plain sequences of characters that are stored in Java strings. Thus the program needs at first to reinterpret these strings into appropriate numbers (integers or reals), prior to assigning them to variables. To parse a string and get its equivalent integer (`int`), one uses `Integer.parseInt(stringname)`; For reals, to parse and create the corresponding float or double from a given string `str`, use the following functions: `Float.parseFloat(str)` or `Double.parseDouble(str)`. Let us revisit Euclid's GCD program by taking the two numbers a and b from the program arguments:

```
class gcd {
public static void main(String[] arg)
{
// Parse arguments into integer parameters
int a= Integer.parseInt(arg[0]);
int b= Integer.parseInt(arg[1]);
System.out.println("Computing GCD("+a+", "+b+"");

while (a!=b)
{
if (a>b) a=a-b;
else b=b-a;
}
// Display to console
System.out.println("Greatest common divisor is "+a);
}
}
```

Compiling and running this program yields:

```
prompt%java gcd 234652 3456222
Computing GCD(234652,3456222)
Greatest common divisor is 22
```

But there is more. In Chapter 1.8, we explained the basic mechanism of input/output redirections (I/O redirections). Using I/O redirections with program arguments yields an efficient framework for executing and testing programs. Let us export the result to a text file named `result.txt`:

```
prompt%java gcd 234652 3456222 >result.txt
```

Then we saved the texts previously written on the console to that file. We can visualize its contents as follows:

```
prompt%more result.txt
Computing GCD(234652,3456222)
Greatest common divisor is 22
```

We are now ready to proceed to the next chapter concentrating on functions and procedures.

2.8 Exercises

Exercise 2.1 (Integer parity)

Write a program that interactively asks for an integer at the console and reports its odd/even parity. Modify the code so that the program first asks the user how many times it would like to perform parity computations, and then iteratively asks for a number, compute its parity, and repeat until it has performed the required number of parity rounds. Further modify this program so that now both input and output are redirected into text files, say `input.txt` and `output.txt`.

Exercise 2.2 (Leap year)

A leap year is a year with 366 days that has a 29th February in its calendar. Years whose division by 4 equals an integer are leap years except for years that are evenly divisible by 100 unless they are also evenly divisible by 400. Write a program that asks for a year and report on whether it is a leap year or not. Modify this code so that the program keeps asking for years, and compute its leap year property until the user input `-1`.

Exercise 2.3 (Displaying triangles)

Write a program that asks for an integer `n`, and write on the output a triangle as illustrated for the following example (with $n = 5$):

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

Exercise 2.4 (Approximating a function minimum)

Let $f(x) = \sin(\frac{25}{x^2-4x+6}) - \frac{x}{3}$ for $x \in [0, \pi]$ be a given function. Write a program that takes as argument an integer `n` and returns the minimum of $f(x)$ on the range $x \in [0, \pi]$ evenly sampled by steps $\frac{1}{n}$. Execute the program for $n = 10$, $n = 100$ and $n = 100000$ and check that the root is about -1.84318 .

Exercise 2.5 (Computing $\sqrt{a}$ numerically)

Consider Newton's root finding method to compute $\frac{1}{\sqrt{a}}$ by choosing function $f(x) = a - \frac{1}{x^2}$. Show that we obtain the following sequence:

$x_{n+1} = \frac{x_n}{2}(3 - ax_n^2)$. Write the corresponding program. Does it converge faster or slower than Newton's method for $f(x) = a - x^2$?

Exercise 2.6 (Variable scope)

Consider the following program:

```
class ScopeExercise
{
public static void main(String [] a)
{
int j=5;
for (int i=0;i<10;i++)
    System.out.println("i="+i);
j+=i+10;
System.out.println("j="+j);
}
}
```

Explain what is wrong with this program. How do we change the scope of variable i in order to compile?

Exercise 2.7 (Chevalier DeMere and the birth of probability **)

In the 17th century, gambler Chevalier De Méré asked the following question of Blaise Pascal and Pierre de Fermat: How can one compare the following probabilities

- Getting at least one ace in four rolls of a dice,
- Getting at least one double ace using twenty-four rolls of two dices.

Chevalier De Méré thought that the second chance game was better but lost constantly. Using the function `Math.random()` and loop statements, experiment with the chance of winning for each game. After running many trials (say, a million of them), observe that the empirical probability of winning with the first game is higher. Prove that the probability of winning for the first and second games are respectively $\left(\frac{5}{6}\right)^4$ and $\left(\frac{35}{36}\right)^{24}$.

Exercise 2.8 (Saint Petersburg paradox **)

The following game of chance was introduced by Nicolas Bernoulli: A gamer pays a fixed fee to play, and then a fair coin is tossed repeatedly until, say, a tail first appears. This ends the game. The pot starts at 1 euro and is doubled every time a head appears. The gamer wins whatever is in the pot after the game ends. Show that you win 2^{k-1} euros if the coin is tossed k times until the first tail appears. The paradox is that whatever the initial fee, it is worth playing this game. Indeed, prove that the expected gain is $\sum_{k=1}^{\infty} \frac{1}{2^k} 2^{k-1} = \sum_{k=1}^{\infty} \frac{1}{2} = \infty$. Write a program

that simulates this game, and try various initial fees and number of rounds to see whether you are winning or not. (Note that this paradox is mathematically explained by introducing an expected utility theory.)

A Concise and Practical Introduction to Programming
Algorithms in Java

Nielsen, F.

2009, XXVIII, 252 p. 27 illus., Softcover

ISBN: 978-1-84882-338-9