Chapter 6

Reasoning about program behaviour – Assertions

To analyze, verify, and describe program behaviour, it is often helpful to look at the *program states*, i.e., the values of the variables. The user is often only interested in the result of a program or fragment, not in what it does to achieve that. This result is captured by the values of the variabels in the last state of the program or fragment.

6.1 Aim

We want to reason about program behaviour.

6.2 Means

An *assertion* is a description of the program state in terms of the program variables. It can be a sentence in English or a logical predicate. An assertion is always associated to a program location. Its meaning is that whenever program execution reaches that location, the assertion should hold. In general, an assertion is included in the program as a line of comment. For example:

```
//assert x >= y
```

When the assertion is a legal Java boolean expression, the comment sign can be left out (don't forget the semicolon):

```
assert x >= y;
```

Every time this statement is reached, the boolean expression is evaluated. When it is true, execution continues, but if it is false, execution is stopped and an error message is issued. (In fact, an *exception* is thrown, see chapter 26 for more information on this topic.)

Such an assert statement can also be used for debugging. A String can be added to the statement, after a colon (`:`), that will be displayed together with the error message. Example:

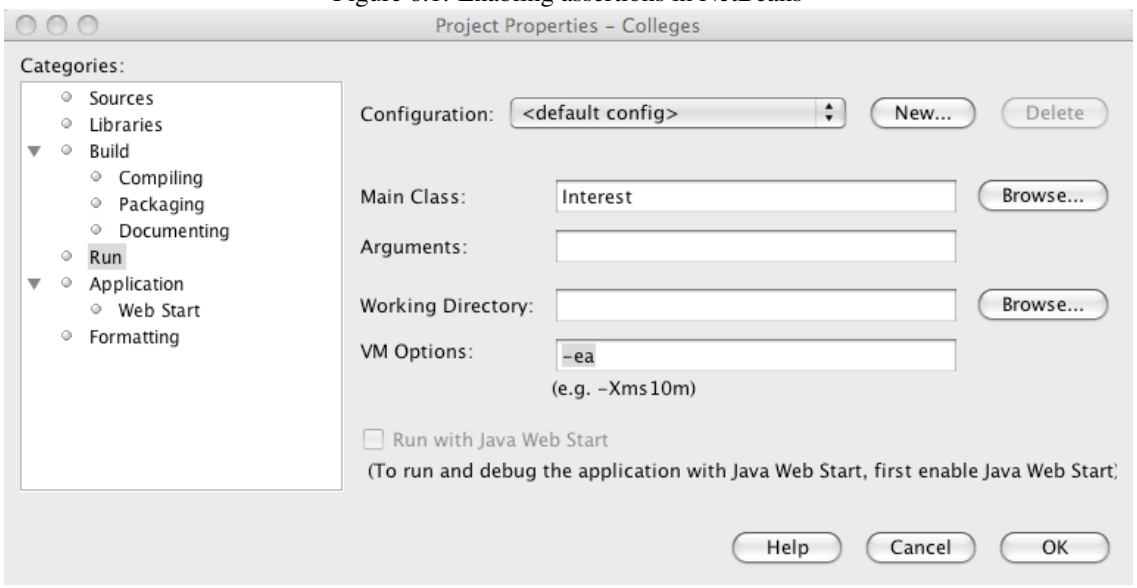
```
assert x>0 :  "x should be positive, but is " + x;
```

Suppose for example that $x = -1$ when this statement is executed, the following message will appear. The number 4 refers to the line number where this statement occurs in the source file.

```
Exception in thread "main" java.lang.AssertionError:  x should be positive, but
is 0 at Ass2.main(Ass2.java:4)
```

Remark The assert statement is by default disabled in Java, which means that no error messages are generated even when an assertion is false. To enable assertions, the execution option *ea* has to be turned on. In NetBeans, it can be specified as an option to the VM (see figure 6.1).

Figure 6.1: Enabling assertions in NetBeans



6.2.1 Example

The following class has a method calculateAbsolute that calculates the absolute value $|x|$ of x and stores it in y.

```
1  class Abs {
2      int x;
3      int y;
4
5      void calculateAbsolute {
6          Scanner scanner = new Scanner( System.in );
7          x = scanner.readInt();
8
9          if (x>0) {
10             //assert x > 0;
11             y = x;
12             //assert x > 0 && y == x;
13             //assert y == |x|;
14         } else {
15             //assert x <= 0;
16             y = −x;
17             //assert x <= 0 && y == −x;
18             //assert y == |x|;
19         }
20         //assert y == |x|;
21     }
22 }
```

The assertions capture the relevant information about the execution. After the assignment `y := x` in line 11, we know that `y == x` and still `x > 0` (assertion of line 12). Hence `y==|x|`, by definition of the absolute operator. So the assertion of line 13 is a consequence of that in line 12, which is expressed by putting the assertions in this order. In the else-branch we have a similar situation and we see that at the end of both branches we have `y == |x|` (assertion 13 and 18). So we know that after the complete if-statement, this assertion should also hold, which is expressed in assertion 20. This assertion is also the desired result. of the method. We call such an assertion the *post-condition* of a method or program.

Chapter 7

Indexed lists of variables – Arrays of variables

Up till now only the built-in types were used. To enable structured storage there is a type constructor in Java to make new types from already available ones.

7.1 Aim

We want to store several data items of the same type in a list that can be accessed using indices.

7.2 Means

We use the *array* construct to make a new type that is a list of fixed length of variables of an already available type, with integer indices. To use such a type involves three things.

1. Definition of the new type, an array of the desired type. Java provides variables of the new array type if `[]` is put behind the type of the desired items (see the example).
2. Declaration of a variable of the new type. (Step 1 and 2 are performed together.)
3. Creation of an array of the desired type. Java provides the new array type if `[]` is put behind the type of the desired items. The length, i.e., the number of elements, of the array is set here, by putting it between the `[]`. After creation the length of an array cannot be changed anymore.

The indexing starts with 0. Elements of an array are accessed by writing the array name and the index, say *i*: ⟨*arrayname*⟩`[`*i*`]`. Each element is a variable of the given type, so all operations of that type can be performed on it, and also assignment.

Provided with an array is the variable ⟨`arrayname`⟩`.length`, that contains the length of the array. Note, that since the indexing starts at 0, the last index in the array is the length minus 1.

As an example we write a program that stores numbers of type `int` in an array `numbers`.

```
import java.util.*;

class ArrayDemo {
  // handles array of integer numbers
  Scanner sc = new Scanner(System.in);

  int[] numbers; // introduction of the type int[]
                                                    // and declaration of the array numbers
  // fills and prints array
  void demo(){
    numbers = new int[5]; // creation of the array numbers

    for (int i = 0; i < numbers.length; i++) {
      System.out.println("Type array element with index " + i);
      numbers[i] = sc.nextInt(); // initialization of the elements of numbers
    }

    System.out.println("First array element is " + numbers[0]);
    System.out.println("Array element with index 2 is " + numbers[2]);
    System.out.println("Last array element is " + numbers[numbers.length - 1]);
  }

  public static void main(String[] args) {
    new ArrayDemo().demo();
  }
}
```

NB In the example we put the creation of the array in the method. It is also possible, to create the array outside the method, when the variable is declared: `int[] numbers = new int[10];` Java does not allow to do declaration and creation as separate steps outside the method.

Arrays of more dimensions are also possible in Java: for example, a 2-dimensional array of integers is defined as an array of arrays through `int[][]`. Creation then is split over several statements. First, a statement where between the first `[]` pair the number of rows is written, leaving the next `[]` pair empty. Second, statements for each row where between the next `[]` pair the number of columns for that row, i.e., the length of the row-array, is written. Numbering starts from the left-hand upper corner, top to bottom for rows (!), left to right for columns.

Also, two dimensional arrays with rows of unequal length are possible: ragged arrays. The number of row-arrays is obtained through ⟨*arrayname*⟩`.length`, the number of elements in each of the row-arrays `i` through ⟨*arrayname*⟩`[i].length`.

Chapter 8

Procedural composition – Methods

Up till now we described the data manipulation task in a class in just one method. When the task is large, it may be difficult to write, understand and maintain the code for it.

In programming, there are many situations like this, where something is too complex to be dealt with conveniently as a whole. The general approach of *partitioning* can often be applied in such situations. Here a general description of the approach is given, followed by a specific application to the case of methods. Other applications of partitioning are given further on in the book.

Partitioning means dividing something complex into less complex parts that can, to some extent, be treated separately. The approach is based on two ideas. For ease of explanation, the case for two parts is considered.

The first idea is *composition*: making the whole from less complex parts that put together form the whole.

To reduce complexity, making a separate part should be simpler than making the whole directly. This requires that when making one part, not all the details of the other part should have to be considered. Similar holds for putting the parts together. The second idea in the partitioning approach is therefore to make an *abstraction* of a part: a description of a part that provides only the information necessary to make the other part. Similar for putting the parts together. That such an abstraction can be made, depends on choosing relatively independent parts.

The idea of partitioning is now applied to the method.

8.1 Aim

We want to partition data manipulation tasks into subtasks.

8.2 Means

Applying the idea of composition, we declare and define several methods, with different names, in a class, that each contain the description of a subtask. One method is, as usual, started from `main`. This starts the processing; from this method others are started, *called*, these may call other methods, and so on. A method call takes place when control arrives at a statement that consists of the method name followed by a (often empty) pair of parentheses. In this way the parts are combined.

The process of defining methods is called "grouping and naming": grouping of code in a method that has a name by which it can be called. This code can be used repeatedly and at different places in the code by writing a call where desired.

Applying the idea of abstraction, we write a short description of the effect of the method on the instance

variables as a comment just before the method declaration.

The various methods manipulate the data stored in the instance variables of the class. These variables are accessible for all methods: it is the data manipulation that is partitioned, not the data storage.

```
import java.util.*;

// handles account
public class AccountPartitioned1 {
    Scanner sc = new Scanner(System.in);

    double balance;
    int rate;

    // initializes
    void initialize() {
        rate = 5;
        System.out.println("Type amount for balance.");
        balance = sc.nextDouble();
    }

    // shows balance
    void showBalance() {
        System.out.println("Balance is " + balance + ".");
    }

    // updates balance
    void updateBalance() {
        balance = balance + (rate * balance)/100;
        System.out.println(rate +" % interest has been added.");
    }

    // handles balance
    void handleBalance() {
        initialize(); // call to method initialize
        showBalance(); // call to method showBalance
        updateBalance(); // call to method updateBalance
        updateBalance(); // second call to method updateBalance
        showBalance(); // second call to method showBalance
    }

    // start-up code
    public static void main(String[] args) {
        new AccountPartitioned1().handleBalance(); // start method handleBalance
    }
}
```

Output is (input in bold) :

```
Type amount for balance.
60
Balance is 60.0.
5 % interest has been added.
5 % interest has been added.
Balance is 66.15.
```

We declare and define methods in the class, as before, but now several rather than just one. The class has four so-called instance methods, `initialize`, `updateBalance`, `showBalance` and `handleBalance`

(the method `main` is a special case, it is not an instance method). Three of them are called from the method that is executed first, i.e., `handleBalance`. The same method can be called several times, e.g., `updateBalence` is called twice here.

We obtain the description of the whole task as the composition of the description of subtasks, `initialize`, `updateBalance`, `showBalance`. The composition is subtle, in the sense that there is no separate combination operator: we combine the descriptions of the parts through the calls in this case inside one of them, `handleBalance`. So the parts themselves contain the information about their composition.

The abstraction is that to write each of the methods we only have to consider the variables of the class and abstract descriptions of the other methods. We provide the abstract descriptions as a comment above each method that describes its activity. The code then needs not be read. When sensible names are chosen for the methods, often even just these provide enough information.

8.3 Execution model

Example to be provided

The execution model reflects that a method becomes part of the object when it is called. Although the start-up code will be discussed in more detail later, it is mentioned now that in this start-up code the method `handleBalance` is called.

From then on, further calls are made from the methods in the object, like `initialize` being called from `handleBalance` Also the method from which the call is made, here `handleBalance` remains part of the object. When a method is executed, a new *method execution box* is drawn inside the object. The body of the method is displayed inside the box, below the dotted line. The control marker is put at the beginning of the body of the called method. A *passive control marker* is left at the place of the call andr signals the place where control has to return when the called method is finished. After the method that has been called finishes, the method execution box is removed from the object and the passive marker becomes active again, putting control just after the method call.

So the object has a more permanent existence then the methods. The instance variables of the object are pictured outside the methods and are accessible for the methods; they are as persistent as the object is. It is helpful to view the object as a data container, storing values, with methods to manipulate the values.

In the same way as partitioning in methods was used in writing the description of the manipulation task of the class, this partitioning is helpful in understanding the execution. As shown in the example, the existence of methods in an object may vary during execution – this is not directly seen from the class description but it is reflected in the execution model.

Chapter 9

Procedural abstraction – Local variables

An object is a data container, storing data like a bank account balance in instance variables, and manipulating the data by means of various methods. Such data is persistent in that it is preserved over the execution of different methods.

However, additional data may be used in the execution of one method, for example a while-loop counter. Such data is relevant only to individual executions of that specific method rather than needing to be preserved in an instance variable of the object. Such data therefore belongs in the method that uses it rather than in the object and is not accessible from outside the method execution.

Local variables in methods enable better abstraction.

9.1 Aim

We want to describe temporary storage of data that only is required for the duration of a method execution, as opposed to the persistent data that the object is the container for.

9.2 Means

We use *local* (with respect to the method) variables that are declared in the body of the method; usually at the beginning of the method body.

Consider the compound interest program from chapter 5. There the number of iterations depended on a desired amount in the balance to be achieved.

We now consider a variant of the compound interest example where the number of iterations is chosen by the user. The amount in the balance is the relevant persistent data for this class, so it is stored in an instance variable, `balance`.

The number of iterations is of relevance for computing the compound interest only, the method `compoundInterest`. Hence we make this number a local variable, say `years`, of `compoundInterest`.

```
import java.util.*;

// handles account
public class AccountCompoundLocal {
    Scanner sc = new Scanner(System.in);
```

```
    double balance;
    int rate;

    // inputs and sets balance
    void initialize() {
        System.out.println("Provide amount for balance and interest rate.");
        balance = sc.nextDouble();
        rate = sc.nextInt();
    }

    // shows balance
    void showBalance() {
        System.out.println("Balance is " + balance + ".");
    }

    // compute and add compound interest
    void compoundInterest() {
        int years; // local variable, counter for number of iterations

        System.out.println("How many years of compound interest you want?");
        years = sc.nextInt();

        while (years > 0) { // uses the local variable years
            balance = balance + (rate * balance)/100;
            showBalance();
            years = years-1; // updates the guard
        }
    }

    // handles balance
    void handleBalance() {
        initialize();
        compoundInterest();
    }

    // main instantiates and uses the object
    public static void main(String[] args) {
        new AccountCompoundLocal().handleBalance();
    }
}
```

Output of this program is (input in bold):

```
Provide amount for balance.
60
How many years of compound interest you want?
2
Balance is 61.8.
Balance is 63.6 ...
```

The advantage of having a local variable instead of an instance variable, is that its influence on the rest of the code, the so-called *scope* is more limited. Changes in a local variable of one method do not influence other local variables with the same name in other methods. When the programmer designs a method, he doesn't have to take into account which local variables are declared in other methods.

9.3 Execution model

Local variables are drawn at the top of the method execution box, above the dotted line. They are just as instance variables, the only difference being their location.

Example to be added

As can be seen from the execution model, the variable `years` only exists while compoundInterest exists.

9.4 Remark

The for-construct as discussed in chapter 5 allows an even stronger localization of variables than placement in a method rather than in an object.

We first show how the last example changes to a for loop and then discuss the further localization.

```
for (int years = sc.nextInt(); years > 0; years = years−1) {
    balance = balance + (rate ∗ balance)/100;
}
```

The first item of the three items in the control part of the for-loop, initialization of the guard, also allows to declare the guard variable there. So the following further change is possible, moving the declaration of `years` inside the for-construct.

The result of this change is, that as far as the program concerns, this variable `years` exists only for the for loop fragment: if another variable with the same name `years` is declared, it is in fact another variable and it will behave as if it had a different name. The idea is that the variable `years` is relevant for the for loop only, and thus need not be available outside that part of the code. For example when various indexed loops occur, this makes the parts of the code less dependent of each other and simplifies writing and maintaining the code.

Chapter 10

Method interface – Parameters and return value

A method provides flexibility as to where in the code its task is described to be performed: anywhere the method is called. Up till now a method acted on values stored in the instance variables of the object or in local variables of the method. This makes it not so flexible as regards selecting the values with which the method performs the task: these have to be stored in these fixed variables. Often the data manipulation task that a method performs is useful to apply for various values that are not all stored in these fixed places, but that rather are selected at different applications of the method. For example, a method that computes the square root should be able to do so for values stored in many different places or even to values directly given to the method as a literal. Explicitly providing different choices of such values to the method is enabled by the parameter mechanism.

Similarly, the result value of a method might be desired to be made available not in a fixed variable but more flexibly. In case of a square root, it should be possible to make the result available wherever needed. Explicitly providing a result value from a method is enabled by the return mechanism.

The idea to achieve this is to have special local variables in the method that contain the desired start values when the method starts. Then the method executes using these start values. Finally, a result can also be put in a special local variable. Because a method provides abstraction in he sense that local variables are not accessible outside the method, a language feature is introduced to provide values to the method and to obtain the result value.

10.1 Data to methods – input parameters

We show how data can be provided to a method.

10.1.1 Aim

We want to provide data to a method explicitly in a flexible manner.

10.1.2 Means

We provide data to a method through a *parameter*. In the header of the method, inside the parentheses after its name, *formal parameters* and their types are added. A formal parameter can be used in the body of the method as a local variable, for example on the right hand side of an assignment. It can be viewed as a local

variable, with one difference: it is initialized at the call. In the method call, we fill in expressions between the parentheses: the *actual parameters*. When the method is executed, as the first action the value of the actual parameters is assigned to the formal parameters in the body. The number and types of the actual parameters has to match the list of formal parameters exactly. The order in which the actual parameters appear in the call determines to which formal parameters their values are assigned.

In the example below the interest rate is a parameter to the method `addInterest`.

```
import java.util.*;

// handles account
class AccountParameter {
    Scanner sc = new Scanner(System.in);

    double balance;
    int rate;

    // initializes
    void initialize() {
        System.out.println("Type amount for balance.");
        balance = sc.nextDouble();
        System.out.println("Type amount for interest rate.");
        rate = sc.nextInt();
    }

    // shows balance
    void showBalance() {
        System.out.println("Balance is " + balance);
    }

    // adds interest r to balance
    void addInterest(int r) { // r formal parameter
        balance = balance + (r * balance)/100;
    }

    void handleBalance() {
        int bonusRate;

        initialize();

        System.out.println("First, a start interest is addded, at the fixed start rate " + 1);
        addInterest(1); // value 1 as actual parameter
        showBalance();

        System.out.println("Next, a bonus interest is added, at the bonus rate that you type in ");
        bonusRate = sc.nextInt();
        addInterest(bonusRate); // value of 'bonusRate' as actual parameter
        showBalance();

        System.out.println("Next, the interest is added according to the account rate " + rate);
        addInterest(rate); // value of 'rate' as actual parameter
        showBalance();
    }

    //start-up code
    public static void main(String[] args) {
        new AccountParameter().handleBalance();
    }
}
```

Output is (input in bold):

```
Type amount for balance.
60
Type amount for interest rate.
3
First, a start interest is added, at the fixed start rate 1
Balance is 60.6
Next, a bonus interest is added, at the bonus rate that you type in
7
Balance is 64.842
Next, the interest is added according to the account rate 5
Balance is 66.78726
```

10.1.3 Execution model

Parameters are drawn as variable boxes above the dotted line in method executions. To distinguish them from local variables, their name is prefixed with a @-character. When the method execution is created, the parameter boxes are filled with the value of the actual parameters.

10.2 Data from methods – return value

We show how data can be obtained from a method.

10.2.1 Aim

We want to obtain data from a method explicitly in a flexible manner.

10.2.2 Means

We obtain data from a method through the `return` statement of that method. At the end of the method body, a `return` statement is added, consisting of the word `return` followed by an expression that has the desired data value, terminated by a semicolon. We say that the value of this expression is *returned by the method* and the effect is that this value is filled in for the method call in the expression where the call appears. This valueis used in the evaluation of the expression.

The type of the returned value is given at the header of the method definition: instead of `void`, the type is written.

Note that, in contrast to the input case where more than one parameter is possible, only one value can be returned.

In the example below the interest is returned by the method `giveInterest` and can either be displayed to be collected or added to the balance.

Note, that the two mechanisms, parameters and return value, are independent: as shown, a method can have parameters without having a return value and vice versa; a method can also have neither or both.

```
import java.util.*;

// handles account
class AccountReturn {
    Scanner sc = new Scanner(System.in);
```

```
    double balance;
    int rate;

    // initializes
    void initialize() {
        System.out.println("Type amount for balance.");
        balance = sc.nextDouble();

        System.out.println("Type amount for interest rate.");
        rate = sc.nextInt();
    }

    // shows balance
    void showBalance() {
        System.out.println("Balance is " + balance);
    }

    // compute interest at stored rate
    double computeNewBalance() {
        return balance + (rate * balance)/100; // returns balance with interest
    }

    void handleBalance() {
        String choice;

        initialize();

        System.out.println("For balance to be updated with interest, type 'u'.");
        System.out.println("For balance plus interest just to be displayed, type any other letter.");
        choice = sc.next();

        if ( choice.equals("u") ) {
            balance = computeNewBalance();
        } else {
            System.out.println("Balance plus interest would be " + computeNewBalance());
        }

        showBalance();
    }

    //start-up code
    public static void main(String[] args) {
        new AccountReturn().handleBalance();
    }
}
```

Output is (input in bold):

```
Type amount for balance.
60
Type amount for interest rate.
3
Balance to be updated with interest, type 'u'.
Balance plus interest just to be displayed, type any other letter.
u
Balance is 61.8
```

or

```
Type amount for balance.
 60
Type amount for interest rate.
 3
Balance to be updated with interest, type 'u'.
Balance plus interest just to be displayed, type any other letter.
 d
Balance plus interest would be 61.8
Balance is 60.0
```

In the first case, the return value of `computeNewBalance` is assigned to the variable `balance`. In the second case, the return value is put on the console.

10.2.3 Execution model

When a method with return value terminates, the return value is used in the evaluation of the expression in which the call is made. The value is not directly shown in the model, it is indirectly visible in the effect that it has in, e.g., the new value of a variable.

10.3 Functions and side effects

A restricted application of the parameter and return mechanisms enables to write functions.

10.3.1 Aim

We want to program functions.

10.3.2 Means

We use the parameter and return mechanism with additional restrictions.

A function is a special method that has parameters and returns a value. Additionally, a function should have no *side effects*, i.e., should not change any instance variable value.

These features make a function fit to be used in expressions, in a similar fashion as in mathematics.

In the example below, the interest is computed using such a function.

```
import java.util.*;

// handles account
class AccountFunction {
    Scanner sc = new Scanner(System.in);
    double balance;
    int rate;

    // initializes
    void initialize() {
        System.out.println("Type amount for balance.");
        balance = sc.nextDouble();
    }

    // shows balance
```

```
    void showBalance() {
        System.out.println("Balance is " + balance);
    }

    // returns interest for balance amount b at rate r
    double computeInterest(double b, int r) { // b and r formal parameter
        return (r * b)/100;
    }

    void handleBalance() {

        System.out.println("For a balance under 1000, the interest rate is 3%.");
        System.out.println("For example, 900 yields interest " + computeInterest(900, 3));

        System.out.println("For a balance over 1000, the interest rate is 4%.");
        System.out.println("For example, 1100 yields interest " + computeInterest(1100, 4));

        System.out.println("Please type your choice for balance amount.");
        System.out.println("We will add a year of interest.");

        initialize();

        if (balance < 1000) {
            rate = 3;
        }else{
            rate = 4;
        }
        balance = balance + computeInterest(balance, rate);

        showBalance();
    }

    //start-up code
    public static void main(String[] args) {
        new AccountFunction().handleBalance();
    }
}
```

Output is:

```
For a balance under 1000, the interest rate is 3%.
For example, 900 yields interest 27.0
For a balance over 1000, the interest rate is 4%.
For example, 1100 yields interest 44.0
Please type your choice for balance amount.
We will add a year of interest.
Type amount for balance.
 1100
Balance is 1144.0
```

10.3.3 Execution model

As follows from combining the paramaters and return values.

Chapter 11

Procedural – Method recursion

Some data manipulation tasks can, by their nature, best be performed as performing some small manipulation task on the result of first performing the same task on simpler data. This is repeated until such a simple task results that it can be performed directly. The small manipulation tasks created in this process are then performed, in the order opposite of the one in which they were created.

This is the *recursive* (recurring to itself) execution of a task: a recursive algorithm. A task is called recursive by nature if a recursive algorithm is the easiest kind of algorithm to design for it.

11.1 Aim

We want to perform a task recursively (the simplification usually being in terms of the data).

11.2 Means

We write a *recursive* method for the task. The method body firstly contains code that describes the recursive part of the task, followed by a call to the method itself, with simpler data. This part is executed when the simplest case is not reached yet. Secondly, the method body contains code that describes how to directly perform the task for the simplest case. This code is executed when the method is called with data corresponding to the simplest case.

The data that is relevant for the recursive execution of the task, and thus has to become simpler at each recursive call, is usually provided through parameters. To enable use of the results of the calls, in building the total result, the results of the calls are usually provided through the return.

NB A recursively performed task can always also be performed using a, different, iterative algorithm. Translation often is not easy. Usually an iterative algorithm is more efficient, especially in terms of memory use, than a recursive one.

In the example below, a recursive method is used to compute the factorial of an integer number.

```
import java.util.*;

class FactorialParameterReturn{
  Scanner sc = new Scanner(System.in);

  long factorial(int n){
    if (n==0)
        return 1;
```

```
    else
        return n*factorial(n-1);
    }

  public void demo(){
    int n;

    System.out.println("Provide a number to compute the factorial for");
    n = sc.nextInt();
    System.out.println("Factorial of " + n + " is " + factorial(n));
    }

// main instantiates and uses the object
  public static void main(String[] args) {
    new FactorialParameterReturn().demo();
  }
}
```

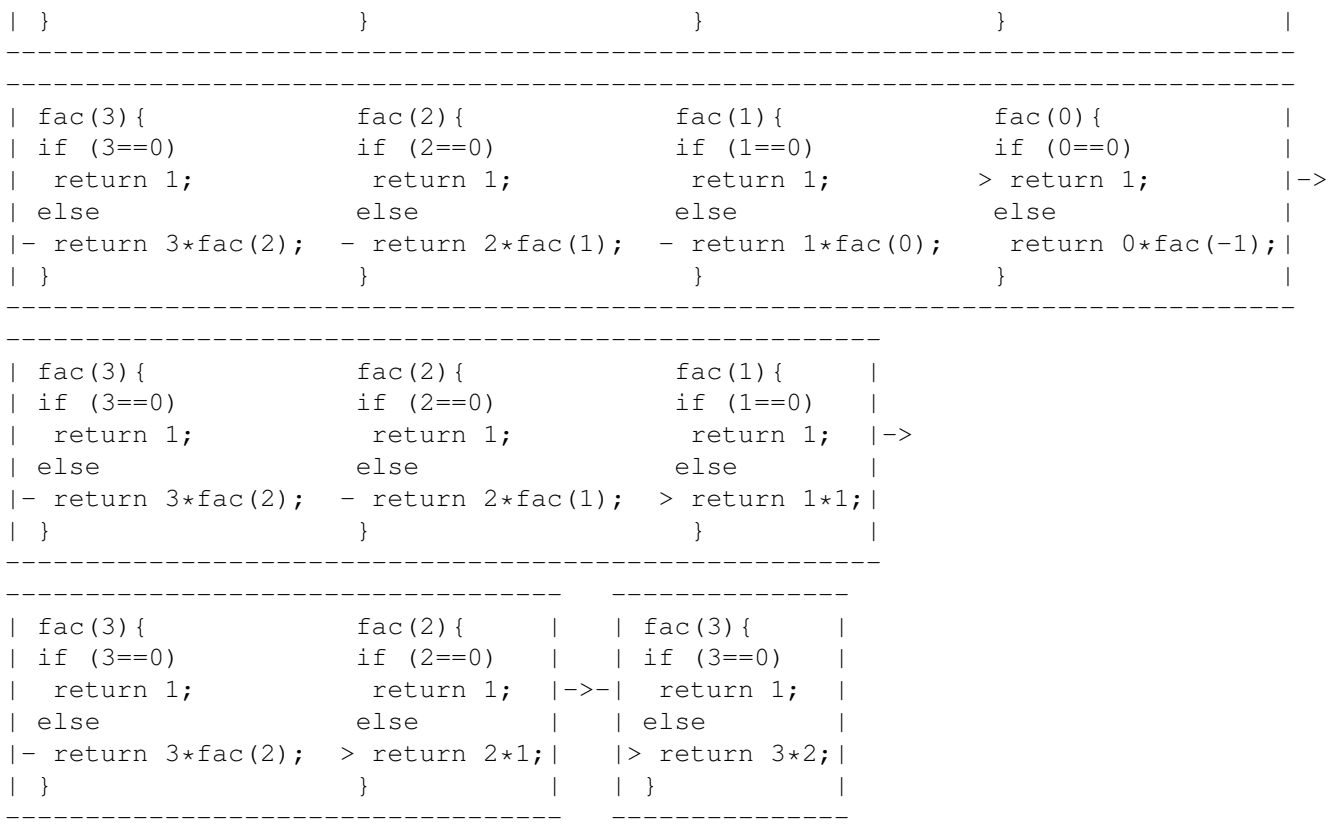
An alternative is, to not store each intermediate values in a separate variable, as is the case in the above approach using parameters and returns, but re-use the variables at each step. This involves undoing value changes as in the following adaptation of factorial.

```
import java.util.*;

class Factorial{
  Scanner sc = new Scanner(System.in);

  int n;
  long fac;

  void factorial(){
    if (n==0)
        fac = 1;
    else{
        n=n-1;
        factorial();
        n=n+1;
        fac=n*fac;
    }
  }

  public void demo(){

    System.out.println("Provide a number to compute the factorial for");
    n = sc.nextInt();
    factorial();
    System.out.println("Factorial of " + n + " is " + fac);
    }

// main instantiates and uses the object
  public static void main(String[] args) {
    new Factorial().demo();
  }
}
```

11.3 Execution model

Each recursive call gives rise to a new instantiation of the method, until the simplest case is reached. When, starting with the simplest case, the execution of some method is completed, the result is returned to the previous method and the returning method disappears.

```
---------------------   --------------------
|>fac(3){            |   | fac(3){          |
| if (n==0)          |   | if (3==0)        |
|  return 1;         |->-|  return 1;       |->
| else               |   | else             |
|  return n*fac(n-1);|   |> return 3*fac(2);|
| }                  |   | }                |
---------------------   --------------------
----------------------------------------
| fac(3){            >fac(2){           |
| if (3==0)           if (n==0)         |
|  return 1;           return 1;        |->
| else                else               |
|- return 3*fac(2);    return n*fac(n-1);|
| }                   }                  |
----------------------------------------
---------------------------------------
| fac(3){             fac(2){          |
| if (3==0)           if (2==0)        |
|  return 1;           return 1;       |->
| else                else             |
|- return 3*fac(2);  > return 2*fac(1);|
| }                   }                |
---------------------------------------
-------------------------------------------------------------
| fac(3){             fac(2){           >fac(1){          |
| if (3==0)           if (2==0)          if (1==0)        |
|  return 1;           return 1;          return 1;       |->
| else                else               else             |
|- return 3*fac(2);  - return 2*fac(1);  - return 1*fac(0);|
| }                   }                  }                |
-------------------------------------------------------------
-------------------------------------------------------------
| fac(3){             fac(2){             fac(1){          |
| if (3==0)           if (2==0)           if (1==0)        |
|  return 1;           return 1;          return 1;       |->
| else                else                else             |
|- return 3*fac(2);  - return 2*fac(1);  > return 1*fac(0);|
| }                   }                   }                |
-------------------------------------------------------------
-----------------------------------------------------------------------------
| fac(3){             fac(2){             fac(1){            >fac(0){          |
| if (3==0)           if (2==0)           if (1==0)           if (0==0)        |
|  return 1;           return 1;          return 1;           return 1;       |->
| else                else                else               else             |
|- return 3*fac(2);  - return 2*fac(1);  - return 1*fac(0);    return 0*fac(-1);|
```



11.4 Backtracking – Undo

In case of solving a problem where at each point in finding the solution different alternatives can be explored, backtracking can be applied. Backtracking amounts to pursuing one alternative at each choice point, and, if an attempt does not lead to the desired result, canceling that attempt and going back to the choice point asbove it, until a solution is found or all alternatives are exhausted.

Backtracking can be applied with parameters, as below.

```
/** Nimm.java − recursive solution of the game of Nimm
 * Rules: two players remove 1 or 2 chips from the heap, taking turns;
 * whoever takes the last chip loses.
 * @author Kees Huizing
 * @date 31 October 2008
 */

public class NimmParameter {
    java.util.Scanner scanner = new java.util.Scanner( System.in );
    int heap; //starting number of chips; always > 0

    /** calculates winning move, if it exists, recursively
     * @return number of chips to be taken, or 0 if there is no winning move
     */
    int winningMove(int h) {
        int move = 0;

        if (h == 1) { //already lost
            return 0; //indicates there is no winning move
```

```
    } else { // heap > 1
        // try all moves
        move = 0;
        for (int m=1; m<3; m++) {
            if (h > m) { // else m illegal
                    if ( winningMove(h−m) == 0 ) {
                    // no win for opponent: good move!
                    move = m;
                    }
            }
        }
        return move;
    }
}

void yourMove() {
    int m;

    System.out.println("What's your move? ");
    m = scanner.nextInt();
    while (m>2 || m>heap || m<1) {
        System.out.println("What's your move? ");
        m = scanner.nextInt();
        if (m>2 || m>heap || m<1) {
            System.out.println("Illegal move");
        }
    }
    heap = heap − m;
}

void play() {
    System.out.println("what's the heap? ");
    heap = scanner.nextInt();
    while (heap < 1) {
        System.out.println("what's the heap? ");
        heap = scanner.nextInt();
    }

    while (heap > 0) {
        System.out.println("Heap is "+heap);
        if (heap == 1) {
            System.out.println("I take 1");
            heap = heap − 1;
            System.out.println("I lose");
        } else {
            int m = winningMove(heap);
            if (m == 0) {
                // no win, make a random move
                m = (int)(Math.random() ∗ 2) + 1;
                System.out.println("I take "+m+" (but you can win)");
                heap = heap − m;
            } else {
                System.out.println("I take "+m);
                heap = heap − m;
            }
            System.out.println("Heap is "+heap);

            yourMove();
```

```
                if (heap == 0) {
                    System.out.println("You lose.");
                }
            }
        }
    }

    public static void main(String[] a) {
        new NimmParameter().play();
    }
}
```

Backtracking can also be combined with the undo approach, as below.

```
/** Nimm.java − recursive solution of the game of Nimm
 * Rules: two players remove 1 or 2 chips from the heap, taking turns;
 * whoever takes the last chip loses.
 * @author Kees Huizing
 * @date 31 October 2008
 */

public class NimmUndo {
    java.util.Scanner scanner = new java.util.Scanner( System.in );
    int heap; // number of chips; always > 0

    /** calculates winning move, if it exists, recursively
     * @return number of chips to be taken, or 0 if there is no wiinig move
     */
    int winningMove() {
        int move = 0;

        if (heap == 1) { //already lost
            return 0; //indicates there is no winning move
        } else if (heap == 2) {
            return 1; // obviously
        } else if (heap == 3) {
            return 2; // ditto
        } else { // heap > 3
            // try all moves
            move = 0;
            for (int m=1; m<3; m++) {
                heap −= m; // take m
                if ( winningMove() == 0 ) {
                    // no win for opponent: good move!
                    move = m;
                }
                heap += m; // undo "take m"
            }
            return move;
        }
    }

    void yourMove() {
        int m;

        System.out.println("What's your move? ");
        m = scanner.nextInt();
        while (m>2 || m>heap || m<1) {
            System.out.println("What's your move? ");
```

```
            m = scanner.nextInt();
            if (m>2 || m>heap || m<1) {
                System.out.println("Illegal move");
            }
        }
        heap -= m;
    }

    void play() {
        System.out.println("what's the heap? ");
        heap = scanner.nextInt();
        while (heap < 1) {
            System.out.println("what's the heap? ");
            heap = scanner.nextInt();
        }

        while (heap > 0) {
            System.out.println("Heap is "+heap);
            if (heap == 1) {
                System.out.println("I take 1");
                heap -= 1;
                System.out.println("I lose");
            } else {
                int m = winningMove();
                if (m == 0) {
                    // no win, make a random move
                    m = (int)(Math.random() * 2) + 1;
                    System.out.println("I take "+m+" (but you can win)");
                    heap -= m;
                } else {
                    System.out.println("I take "+m);
                    heap -= m;
                }
                System.out.println("Heap is "+heap);

                yourMove();
                if (heap == 0) {
                    System.out.println("You lose.");
                }
            }
        }
    }

    public static void main(String[] a) {
        new NimmUndo().play();
    }
}
```

Chapter 13

Name overloading – Method overloading

Method names can be used to indicate similar methods if the methods differ in the parameter lists.

13.1 Aim

Sometimes methods perform essentially the same tasks, but only differ in the kind of information that is provided as input through their parameters. Such methods can be given the same names, using the difference in parameter lists to distinguish between them.

13.2 Means

We use *overloading* of method names, i.e., methods with the same names but with a different list of parameters are regarded as different methods. Note that a difference in the *names* of parameters only does not count as a difference in methods. Only a difference in the list of *types* is what counts, i.e.., a different number of parameters, a different type of a certain parameter, or a different ordering of types.

In the example below, there are three methods with names that differ only in the parameter lists.

```
class AccountOverloader{
    double balance;
    int rate;

    void initialize() {
        balance = 0;
        rate = 0;
    }

    void initialize(double b) {
        balance = b;
        rate = 0;
    }

    void initialize(double b, int r) {
        balance = b;
        rate = r;
    }
```

```
    void showData(){
        System.out.println("Balance is " + balance);
        System.out.println("Interest rate is " + rate);
    }

    void demo(){
        initialize();
        showData();

        initialize(50.0);
        showData();

        initialize(50.0, 5);
        showData();
    }

    // start−up code
    public static void main(String[] args) {
        new AccountOverloader().demo();
    }
}
```

The three calls lead to the execution of three different methods.

13.3 Execution model

In the execution model, the method that is executed as a result of a call is made visible in the object. A different actual parameter list will lead to a different method being executed and hence a different method body will be drawn in the object.

13.4 Remark

– Overloading and narrower or wider types

As discussed in chapter 2, types may be in a relation, e.g., `int` being a narrower type than the wider type `long`. E.g., values of a narrower type can be assigned to variables of a wider type. Similarly, a value of a narrower type can be used in method calls where the formal parameter is of a wider type.

For the selection of an overloaded method, the "best match" is then chosen, i.e., if there is only a method `m(T t)` and a call `m(v)` is made with with a variable `v` of narrower type `S` of `T`, this method will be executed. However, if there is also a method `m(S s)` available, then this will be the method that is executed at the call `m(v)`.

Subtle mixtures of types could occur, e.g., in case of methods with more than one variable. This may lead to calls where it is ambiguous which method to apply. As an example, consider again the type `T` with narrower type `S`. Let `m(T t, S s)` and `m(S s, T t)` be overloaded methods. Let `v` and `w` be variables of type `T`. In the case of a call `m(v, w)` both methods match, neither of them more precise than the other. This is called ambiguous invocation and results in a compilation error. In general, it is not wise to define overloaded methods that differ only in subtle ways like this.

Chapter 47

Predefined generic classes – Java Collections Classes

The standard Java packages provide a lot of useful classes to organize data.

47.1 Aim

Create and use elementary data structures.

47.2 Means

The *Java Collections Classes*, aka Java Collections Framework. The utility package `java.util.*` provides some elementary data structures to store and retrieve objects of the same type (or subtypes thereof). These classes have paramterized types and are therefore type-safe: the programmer is protected against storing data of the wrong type. The structures are flexible and efficient.

The classes are organized under three interfaces: `List`, `Set` and `Map`.

47.2.1 ArrayList

An *ArrayList* gives the functionality of an array (store and retrieve elements by index) and some extras. The most important difference is that the size is not fixed. The ArrayList shrinks and grows with the needs. Furthermore, it has methods to insert elements at arbitrary places and to search for an element.

Important methods of **ArrayList¡E¿** are:

`E get(int i)` returns the element at position *i*;

`void set(int i, E e` changes the element at position *i* to *e*;

`void add(int i, E e` shifts the elements at position *i* and elements after that one place; inserts element *e* at position *i*;

`void add(E e` appends element *e* at the end of the list;

`int size()` returns the number of elements in the list;

Example

```
import java.util.*;

class Account {
        String name;
        double balance;

        Account(String n, double b){
                name = n;
                balance = b;
        }

        void printInfo(){
                System.out.println("For " + name + " balance is " + balance);
        }
}


class ArrayListDemo{
        Account Ac0;
        Account Ac1;
        Account helper;

        List<Account> loa;

        void demo(){

        System.out.println("Two accounts are made.");

        Account Ac0 = new Account("Kees", 1000);
        Account Ac1 = new Account("Ruurd", 1000);

    System.out.println("A list of accounts is made.");

        loa = new ArrayList<Account>();

        System.out.println("The accounts are put in the list.");

        loa.add(Ac0);
        loa.add(Ac1);

    System.out.println("The account at position 1 is shown.");

        helper = loa.get(1);

        helper.printInfo();
        }

        public static void main(String[] args) {
            new ArrayListDemo().demo();
    }
}
```

47.2.2 Storing primitive types

Type parameters can only be class types, not primitive types. Therefore, values of type `int`, `double`, `boolean`, etc. can not be directly stored in collections classes. The solution is to put them in an object that just contains an instance variable of the appropriate type. Java proivdes *wrapper classes* for this purpose. They have the primitive type, written with a capital initial letter, i.e., `Double`, `Boolean`, etc. The wrapper class for `int` is `Integer`.

Conversion from value to object can be done with the constructor, e.g,

```
Integer wrap = new Integer(37);
```

and the other way around with a method:

```
int value = wrap.intValue();
```

Fortunately, this conversion is done by the compiler for you.

Apart from a method to retrieve the value that is stored in the object, the wrapper classes provide some other useful methods concerning the type. See the API for further information.

47.2.3 Map

47.2.4 Set

The interface Set represents a collection of elements without duplicates.

Chapter 48

Visit all elements of a collection – Iterator interface

48.1 Aim

We want to visit of a collection (some data structure containing elements of similar type) without having to know anything about the implementation of the structure.

48.2 Means

The interface *Iterator* provides some methods to visit the elements of a collection. The classes of the Java Collection Classes all provide an Iterator to visit its elements. The Iterator object can be obtained by the method `Iterator iterator()`. Alternatively, the *Iterator for-loop* can be used to visit the elements (see below).

If you are a developer of a collection-like class, you can write an implementation of Iterator.

48.2.1 Methods of Iterator

The interface `Iterator<T>` has the following methods:

`T next()` Gives the next element of the iteration. If there is no next element, because all elements have been visited, it will throw the runtime exception `NoSuchElementException`.

`boolean hasNext()` Returns *true* if there are still elements to visit.

Note that these methods behave the same as the methods in `Scanner` with the same names. In fact, `Scanner` implements the interface `Iterator <String>`.

When the collection supports it, one can also call the method `void remove()` that removes the current element in the iteration from the underlying collection. `Scanner` does not support `remove`.

48.2.2 Iterator for-loop

A convenient way to visit all elements of a collection that supports the Iterator interface is provided by the **Iterator for-loop**. This is a construction similar to the ordinary for-loop that uses an Iterator behind

the scenes. Suppose `someCollection` refers to an instance of the class `SomeCollection<Element>` is a collection that supports Iterator and contains elements of type `Element`. Then a loop that prints each elements of `someCollection` is as follows.

```
for ( Element e: someCollection) {
        System.out.println( e );
}
```

Not only the collection classes provide Iterators. Arrays and Scanner do it as well.

So `scanner` refers to a Scanner object, the code

```
String s;
while ( scanner.hasNext() ) {
        s = scanner.next();
        System.out.println( s );
}
```

one can also write:

```
for ( String s: scanner) {
        System.out.println( s );
}
```

And if `a` is an array of `int`:

```
for (int i=0; i<a.length; i++) {
        System.out.println( a[i] );
}
```

can be replaced by:

```
for ( int n: a) {
        System.out.println( n );
}
```

Note that in these last two for-loops, the variables `i` and `n` have very different roles. the variable `i` is an *index*, used to obtain the elements by `a[i]`, whereas the variable `n` gives you *directly* an element of the array, without the need of an index. This is convenient, but also a limitation. Since an Iterator (or an Iterator for-loop) do not provide an index into the array, loops that need this index for other purposes than for just getting the elements can not be rewritten as an Iterator for-loop.