Programming Language Notations

CHAPTER 8

In the previous chapters we discussed the principles and concepts of programming. Most programs contain computer executable algorithms, i.e. a set of instructions that tell the computer how to do something. In the last fifty years more than thousand programming languages have been developed for general purpose or to serve special needs. The languages have been classified according to the programming paradigm they support. In this chapter we will present notations and paradigms of the most important programming languages.

CHAPTER OBJECTIVES

- To provide an overview of the programming language generations
- To understand the basic notations of procedural languages
- To understand the concepts of object-oriented programming
- To know about the declarative programming and regular expressions
- To know how to use spreadsheet programs

The "mother tongue" of computers is the **machine language**. An instruction is represented (as any information) in the computer as binary code. Each processor family has its own native machine language. However, it would be very cumbersome if humans were forced to "explain" an algorithm in the computer's native language, because every intricate technical detail has to be dealt with. These include

- address a certain memory location
- fetch the content of the memory address
- load the value into the processor's register or stack
- specify how to manipulate the data values in detail

- store the result back into memory

On modern computers these elementary instructions can run to a certain extent interleaved (*pipelined execution*). The exact way to arrange the elementary instructions for pipelining is done in a special programming technique, called **micro-programming**. It has to consider the timing of instruction and make sure that the result of an operation is ready before the next instruction uses it. To benefit from pipelining the programmer has to know which instructions may run in parallel, e.g. address an memory location and push a data value to the register.

Developing today's complex software systems in machine language would likely be impossible even for well trained programmers. The only benefit is clearly that the computer directly "understands" the program.

A first step to better readable programs was to replace the binary code by some mnemonic. For example, the instruction

move 0 to the register #2 (clear R2)

could be expressed in a (hypothetical) machine code as

$1011\,0000\,1101\,1010_{bin}$ or as $B0\,DA_{hex}$

where the first part $1011\,0000_{bin}$ represents the instruction and the second part $1101\,1010_{bin}$ is the target address for that instruction.

In assembly language this might be written as

CLR R2

where *CLR* represents the instruction code $1011\,0000_{bin}$ and $R2$ is the symbolic address for register #2 (here coded as $1101\,1010_{bin} = DA_{hex}$).

The translation from assembly language to machine language is straight forward and done by an **assembler**. It is just a replacement of the mnemonic by the binary code. Therefore, the machine dependence still remains. The mnemonic is only easier to remember.

A more "human friendly" language that abstracts the technical details is called **high-level language**. We call this property of a programming language **machine independent**. It requires a sophisticated **compiler** to translate the program into machine code. One big advantage apart from the more human readable notations is that the program can be compiled in any machine code - if a compiler for that specific processor exists.

Languages of this type are called **third-generation language** as opposed to machine language (*first-generation*) and assembly language (*second-generation*).

In this chapter we will concentrate on third- and higher-generation languages.

Even if the generations have order numbers this does not mean that higher generations have extinct all lower generations. The order number only reflect the level of abstraction of the programming language. All generations still exist, but new programs are mostly developed in high level languages as we try to visualize in figure 8.1. Machine-code programming started with the construction of the first programmable, electromechanical computer

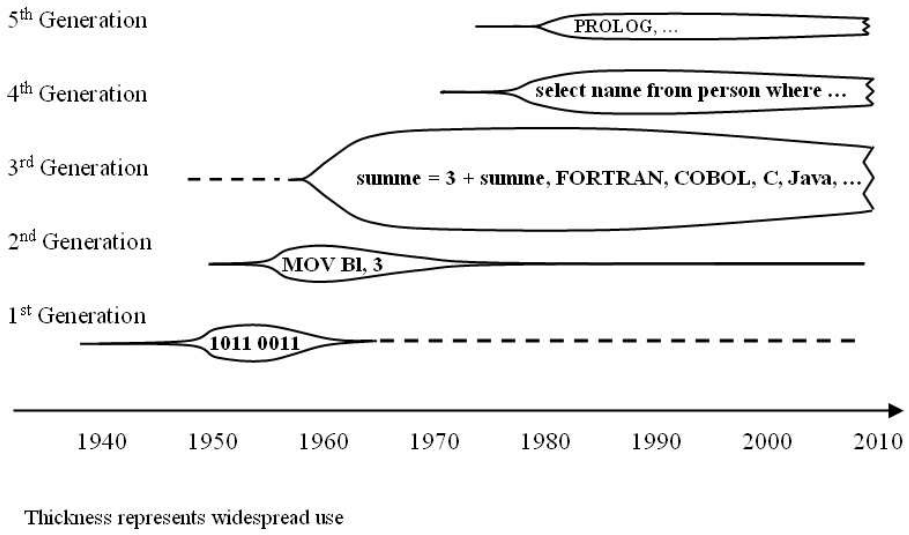


Figure 8.1 Program language generations

Z3 developed 1941 by the German engineer Konrad Zuse. A decade later assembler languages where introduced. The first third-generation language was FORTRAN (FORmula TRANslation) launched officially in 1954. From that time on many other high-level languages have been invented and enhanced with new programming constructs to reflect the state-of-art in programming. The forth-generation languages follow a declarative paradigm that is in contrast to the imperative approach of third-generation languages. Fifth generation languages are designed to state the problem domain only and the computer is expected to solve the problem for you. Those languages (e.g Prolog) are used in Artificial Intelligence mainly.

8.1 Procedural Languages

The *procedural paradigm*, also known as **imperative paradigm**, is closely related to the mathematical formulations of algorithms. A procedural program contains *declaration* of variables and *instruction* that define the program logic. The variables are used to assign or reference data values. The instructions in a program describe precisely how to manipulate the data and how to control the sequence of execution, called the **control flow**. Input and output statements are responsible for the interaction with the user.

Lets look at the declarations, assignment and control statements in more detail now.

8.1.1 Declarations

Algorithms work on data; therefore we need to declare all data that will be used in a program. A data item is referred to by **variable name**. That is a *symbolic address* for that data item. Usually the variable is limited to reference

only data of a declared type. This allows the compiler to know how to interpret the data value and in a case of data assignment the compatibility of types can be checked at compile time. This is known as **static type checking**. Here are some example data declarations in Java and C language:

Java	**C**
final int taxPercent = 18;	#define taxPercent 18;
float netPrice, grossPrice;	float netPrice, grossPrice;
int[ ] salesHistory[10];	int salesHistory[10];

The data types **int** and **float** are language specific and denote integer numbers and floating point numbers (see chap 9). The word **final** denotes in Java a *literal*, i.e. a name representing an constant value. The *salesHistory* is an array of integers with 10 elements where the indexes are numbered 0 to 9.

All third-generation languages (3GL) support numerical, alphanumerical, and boolean data types, only the names for it differ. In addition they support data structures like arrays, records (often called **struct**), and sometimes an union type.

8.1.2 Assignment statements

A value can be assigned to a variable like in Mathematics:

```
netPrice = 98.90;
brutto = netPrice * taxPercent / 100;
salesHistory = salesHistory + 1;
```

The last example clearly indicates that we have an assignment rather than an equation. So, a better notation would be ":=" but most popular programming languages prefer to write the equal sign "=" for an assignment and use "==" for the comparison of two expressions.

8.1.3 Control statements

Statements are executed in the sequence they appear in the program unless there is a control statement. *Control statements* alter the execution sequence of a program. This can be done with a simple *goto*-statement. On the machine level this is nothing more than a JUMP instruction that sets the program pointer to an address other than the next value in sequence.

The most prominent control statement in 3GL languages is the *if-then-else*-statement. It executes a branch depending on a logical expression. If the expression is **true** the statements after *then* are executed; if the expression evaluates to **false** the statements after *else* are executed. The example program flow chart in Figure 8.2 visualizes the control flow of an *if-then-else*-statement depending on the value of the expression *expr*.

The flow chart is drawn in the graphical notation of the Unified Modeling Language (UML) to describe the program control flow. Only three graphical icons and one textual expression are used:

- *rectangle with convex arcs* to represent an action statement

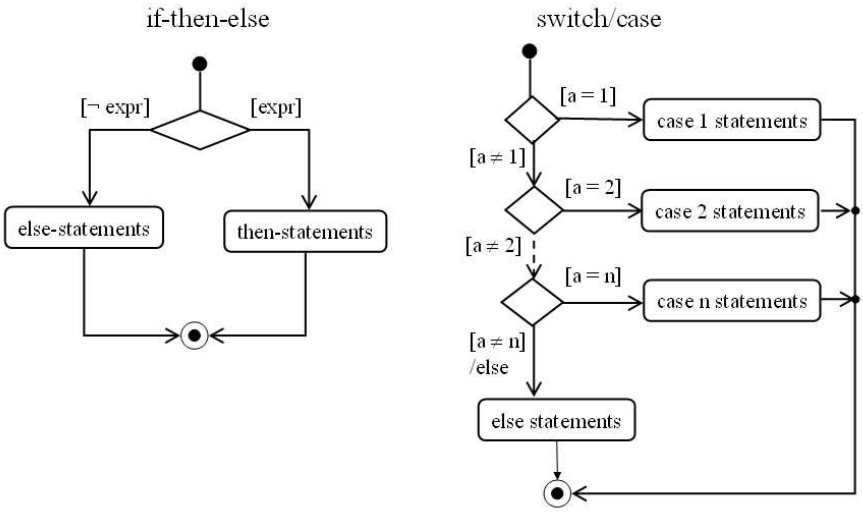


Figure 8.2 Program flow with different control statements

- *diamond* to represent a decision
- *arrow line* to indicate the sequence of two actions
- *logical expression in brackets* to defines the condition for a transition from one action to an other

A multibranching statement is the *switch* or *case* statement that has *n* exits depending on the value of a control variable. This statement can be simulated as cascading if-then-else statements (see example in figure 8.2). The first branch condition that evaluates to **true** decides the statements that are executed. In our example this depends on the value of the variable *a*.

8.1.4 Loops

A loop can be constructed by an *if-then-else* branch statement and an *goto* that jumps back to the branch statement. Depending on the loop condition different names are used:

for-do , if the loop is executed a predefined number of times

while-do , if the check is done prior to the execution of the loop

do-until , if the check is done after to the execution of the loop.

All three mechanisms are shown as a program flow chart in figure 8.3. Please remember that in UML the decision statement is drawn as a diamond (rhomb shape) and the decision criteria is put in brackets and stated as an expression that evaluates to either **true** or **false**. The control follows the arrow line with the condition that yields **true**.

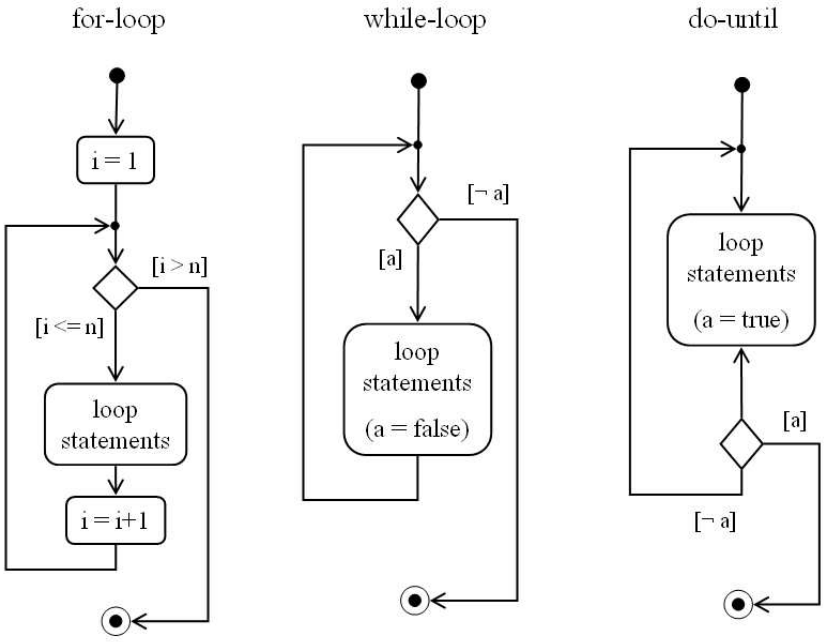


Figure 8.3 Flow charts for program loops using UML activity diagrams

The "for-loop" starts with the action to assign an initial value for the loop-variable i. Then a decision is taken depending on the value of the loop variable i. If the value for i is less than or equal to the upper limit, the left arrow is followed and the loop-statements are executed. The next step is incrementing the loop-variable and we return to the decision statement. Then the loop begins again. If the upper limit of the loop variable i is exceeded, the loop will terminate.

A programming example using a "for-loop" is shown in the last lines of Figure 8.9. The loop variable is iterator object i, the loop condition is the result of the the statement $i.hasNext()$ that returns **true** as long as the iterator i has more elements. The increment of the variable i is done as side effect of the $next()$ method.

The "while-loop" and "do-until" are similar. The "while-loop" checks if the loop criteria is **true** before the execution of the statements. In the case of the "do-until" the check is done after the statements. To avoid an infinite loop it is necessary that the value of the expression is eventually changed during the execution of the loop statements. Please note that termination logic is different for both loops: The "while-loop" terminates on false condition but the "do-until" terminates on true condition.

8.2 Object-Oriented Languages

With the deeper understanding of information hiding and modularization (see chapter 7) the object-oriented paradigm was develop in the 70^{th} and as a result

new programming languages like Modula, Smalltalk, and C++ appeared in the early 80^{th}.

Following the *object-oriented paradigm* we perceive the world and their programmed representation as interacting objects. An **object** is a self-contained entity that consists of both data (called **state**) and operations (called **methods**) to manipulate the data. Objects interact with each other by sending *messages* that invoke the execution of a method at the receiving object.

This concept fits perfect to simulate autonomous processes or machines. No wonder that the first object-oriented language Simula-67 was designed for exactly that purpose. But this "view of the world" is also possible for more general situations.

As example let us take the administration of students. We consider the students as objects. Each student object contains data that represents the state of the student. It is possible to change the state by sending a message to the object. In response to the message the receiving object executes operations on the object's data. So changing a students record is a request (message) to change the data of that object. An new student record corresponds to the creation of a new object. To produce a list of a student's examination results just ask the student's object to do so. It will send a message with the information to the printer object who will render the information on paper.

Sending a message to an object causes the execution of a method. A *method* is similar to a function in procedural languages. Nevertheless, the conceptual difference between both is that the caller of a function decides which function is activated but in object-oriented environments the receiver of a message decides which activities are accomplished. Thus objects can react differently to a message depending on the object's type or state (see also Subsection 8.2.4).

We will now explain the main properties of object orientation.

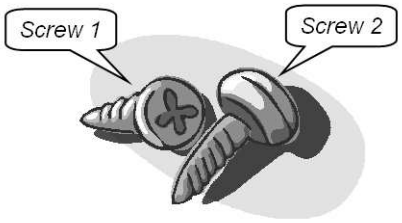
8.2.1 Object identity

Every object has a distinct *identity* (abbr. ID). The identity is unique and eternal. If an object is destroyed an other object cannot take its identity. This postulation is weakened for practical use. For transient objects a memory address may serve as identity because two objects can never share the same memory location. Then, however, the relocation of an object is not possible.

Most object-oriented languages automatically generate a unique binary value (called *surrogate*) that serves as identity (ID) when an object is created. This surrogate makes the object independent of its location (memory or disc address) or state.

But it is possible that two objects are equal, i.e. some of its properties are the same. Which properties are used to define equality depends on the purpose. Two screws might be considered equal if their length, thread, and head are the same. However, for other usage also its hardness and color might be of relevance to consider it as equal. In fact the programmer has to define what he means by *equal*.

Even if two objects are equal in every aspect, it is always possible to distinguish both by its identity. As consequence we have to distinguish between *identity* (ID) and *equality*.



– Java code for
– 2 screws that are equal

```
Screw s1 = new Screw();
Screw s2 = new Screw();
s1 == s2   // false
s1.equals(s2) //true
```

Figure 8.4 Java example: testing for identity and equality

In Java equality is defined with the method *equals()*. To check if two objects *s*1 and *s*2 are equal the message *equals(s2)* with object *s*2 as parameter is sent to the other object *s*1 as in Figure 8.4.

8.2.2 Class concept and hierarchy

Objects with the same behavior are said to be of the same *type*. A *class* is the implementation of a type, i.e. a class defines the internal structure of a type and implements its behavior in form of methods.

Some people call the class an "object factory" because special methods of the class (*constructors*) can create (instantiate) new objects like in line 1 of Figure 8.4. The statement *newScrew*(); creates a new object of class *Screw*.

We can build a new class on the basis of other classes. In this case the new class "inherits" the structure and methods from the base classes. If the *subclass* inherits from more than one class (*superclass*) we call this *multiple inheritance* to contrast with *single inheritance* where each class has at most one superclass. Subclassing provides an elegant way of extending a system without changing existing code.

The power of object-orientation comes from the possibility of reusing existing classes (code).

Most object-oriented languages provide only single inheritance to avoid ambiguity. With single inheritance the classes form an "inheritance tree" where all subclasses inherit from its parent and successively from all ancestor classes. The base class of all classes is called *root class*. Figure 8.5 shows the class hierarchy starting from the root class for person objects. Essential properties (identity, class name, number of objects referencing it) and methods (clone object, represent object as string, standard error handling) are implemented in the root class and propagated to all descendants. Subclasses can provide their own functionality by adding new methods or overriding inherited methods. For instance the income of a manager is calculated differently to an ordinary employee.

The Java code example illustrates the subclassing of class *Employee* from class *Person* in Figure 8.6. The class *Person* implicitly inherits from class *Object* and class *Employee* can use all methods of its ancestors. This is natural as an employee is a person and an object. An employee however has additional properties like a salary and a social security number (SSN). Please note, that

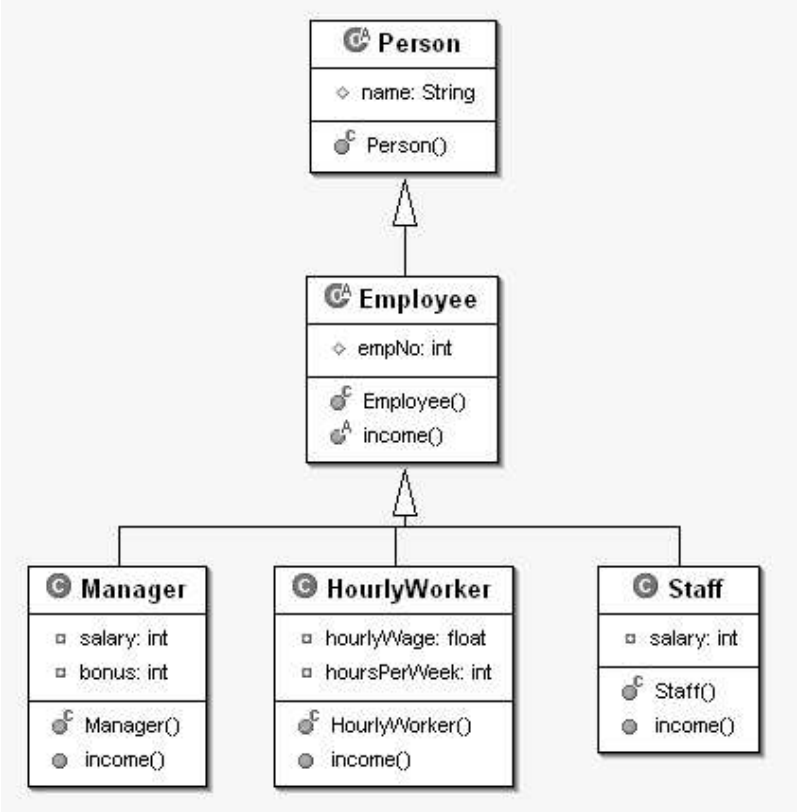


Figure 8.5 Class hierarchy of persons

we use methods to read or set the salary or SSN. In the following section we will look at the consequences of this indirect access in more detail.

8.2.3 Encapsulation and abstraction

Programming objects are considered to be *abstractions* of the real world insofar as only relevant and needed properties or states are modeled and exhibited.

How the properties are implemented and how the state is kept in the object's variables is hidden from the user. It is therefore not possible to change the state of an object directly by assigning a new value. The only way is to do it indirectly by activating a method that changes the value of a variable (see Figure 8.7). This allows the freedom to change the structure or implementation of methods of an object without affecting other objects. The other objects only rely on the interface to remain stable. This concept of hiding structure and method implementation is known as *encapsulation* or "information hiding" as we called it in 7. We prefer the first term because the second is a misnomer: not the information is hidden, but its implementation.

In some languages the object variables are only hidden if this is specified by the word **private** (see Figure 8.6).

```
public abstract class Person /*implicitly extends class Object*/ {

      protected String name;

      public Person (String n) {
            super();
            name = n;
      }

public abstract class Employee extends Person {

      protected int empNo;
      public Employee(String n, int no) {
            super(n);
            empNo = no;
      }

      public abstract int income();

}

public class Manager extends Employee {

      private int salary;
      private int bonus;
      public Manager(String n, int no, int monthlySal, int incentive) {
            super(n, no);
            empNo = no;
            salary = monthlySal;
            bonus = incentive;
      }

      public int income() {
            return salary*12 + bonus;
      }
      …
}
```

Figure 8.6 Java example code for subclassing

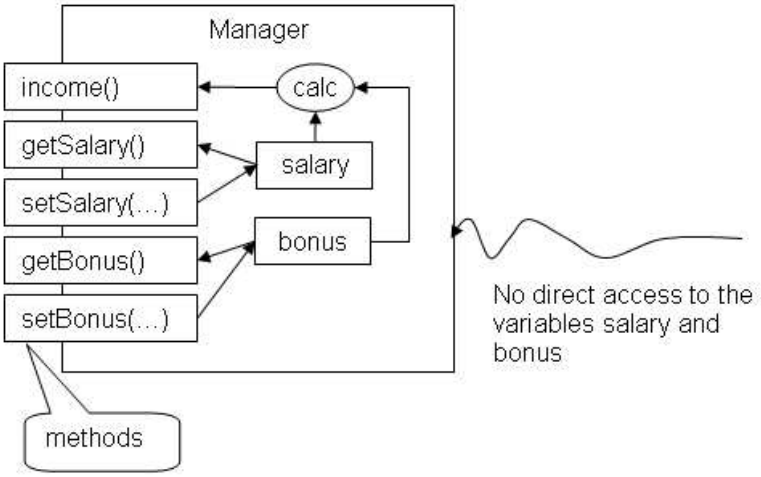


Figure 8.7 Encapsulation of variables salary and bonus in class *Manager*

8.2.4 Polymorphism

In subsection 8.2.2 we learned that a subclass can re-implement a method. As consequence the same message can activate different code and therefore objects of different classes can react differently to the same message.

The idea can be extended to objects of any class, not only for sub- and superclasses. When objects from different classes behave to the same message class-specific, this is called *polymorphism* (from the Greek πωλι = many, and μωρφε = shape).

Polymorphic behavior is not bound to object-oriented languages. A kind of compile-time ("static") polymorphism is used for arithmetic operations in most procedural languages for mixed operands. In fact, behind the scene the addition (+-operation) has four implementations for the two data types int and $float$, one for each combination of the data types:

int	+	int
int	+	float
float	+	int
float	+	float

If there would be no polymorphism we would need **four** different "plus" operation and have to distinguish the possible situations with "if-then-else" statements.

This shows that polymorphism reduces the need for control statements and the number of code lines.

Lets give an other example for polymorphic behavior where the appropriate method can only be decided at runtime. Assume we have an employee class hierarchy as show in Figure 8.8. The yearly income of each employee type is different. The message *income()* will invoke the specific method implementation depending on the receiving object. Please refer to Figure 8.6 for an example how to implement the method *income()*.

To produce a list of all employees with its yearly income the code fragment of Figure 8.9 has to be executed. It is not necessary to know the exact definition of the collection *allEmployees* or understand the syntax completely. For the moment it is sufficient to know, that we can add our employees to that collection and we can use an iterator object to loop over all employees.

Please note, that we do not use control statement like:

if (emp is Manager) then {compute yearly income as $12 * salary + bonus$}
else if (emp is Staff) then {compute yearly income as $13 * salary$}
else if (emp is HourlyWorker) then {compute . . . }

to get the individual income of each employee.

Pure object oriented languages even avoid "loops" with the help of iteration methods and code-block objects. The trick to avoid a loop is that we have an iterator method that takes a code block as parameter. The code-block is applied to each object contained in the receiver.

The loop appearing in the last lines of Figure 8.9 would then be replaced by a single message. In Smalltalk it looks like this:

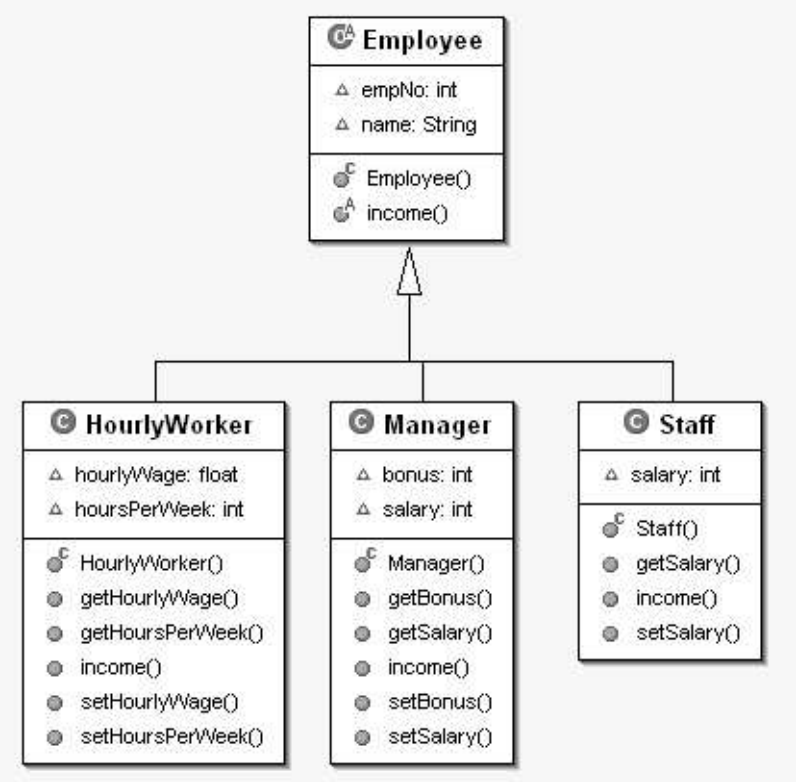


Figure 8.8 UML diagram for employee classes

```
public static void main(String[] args) {
  // Test of polymorphism using employees with different income calculation
  AllEmployees allEmployees = new AllEmployees();
  allEmployees.add(new Staff("Jonathan P. Staff",101, 4000));
  allEmployees.add(new Manager("George W. Manager",102, 5000, 10000));
  allEmployees.add(new HourlyWorker("Tommy Bob Worker",103, 12.5f, 40));
  System.out.println("Name of Employee    yearly Income");
  for (Iterator i = allEmployees.iterator(); i.hasNext(); ) {
    Employee emp = (Employee) i.next();
    System.out.println(emp.getName()+"    "+ emp.income()); // polymorphism
  }
}
```

Figure 8.9 Use of polymorphism for income list of different employee types

```
allEmployees do: [ :emp | emp income printOn: Transcript ]
```

Don't worry about the syntax, we will explain it as far as necessary, now.

In Smalltalk everything is an object and a space instead of a dot is used to separate the receiving object from the message, i.e instead of *receiverOb-*

ject.message() we write *receiverObject message*. The result of a message is always an object, that's why messages can be cascaded. Variables are only declared, no type is assigned to it. In our example the variable *emp* is used to reference the objects of the receiver collection one-by-one. The object referenced by the variable *emp* carries the type not the variable. Messages with a parameter are written in the form "message: parameter". This is the case for the method *printOn:* that has one parameter, the global object *Transcript*.

Assume we already have a collection object *allEmployees* that contains all employees, no matter if they are *Staff, Manager*s, or *HourlyWorker*s. Object *allEmployees* receives the message *do:* with a code block as parameter. The code block is contained in brackets ([. . .]). The receiver collection *allEmployees* executes this block for all its employee objects which are referred to as *emp*. So, each employee object *emp* receives the *income* message and its result is then printed on the *Transcript* which acts as standard output in Smalltalk. The receiver object *emp* depending on its class decides which method implementation to use for *income*. This avoids the use of "if-then-else" or "case" statements.

8.3 Declarative Languages

The programming languages considered so far require that the programmer exactly explains **how** to solve a problem. A *declarative* programming language is a language for describing a problem or specifying **what** is wanted rather than defining an algorithm for the solution.

With declarative programming we only describe the goal which is in contrast to procedural programming languages, where a sequence of operations (imperatives) are given that tell the computer: "do this, take that'".

The distinctions are never sharp: any computer language has declarative elements like variables that represent storage locations, but usually the programmer does not care to know the address or implementation of the data structure.

A higher abstraction level makes the program more independent from the hardware. The higher the abstraction level the lesser implementation details are exhibited. But, sometimes it is important to understand the implementation rather than enjoy the abstraction, e.g. for optimization reason.

As example of a declarative language we present the *Structured Query Language (SQL)*. It is not a general programming language as it works on relations only. It serves as "lingua franca" for relational database systems. *Relations* are similar to the tables of a spreadsheet. More precisely, a relations is a set of equally structured records (see Chapter 4). With SQL you can manipulate (add, delete, update) and retrieve records which are called *tuples* in the concept of relational theory. This is best explained with some examples.

Let *Persons* be a table of persons with four data columns as in Figure 8.10. To add a new person we use the syntax:

```
insert into Persons
values ('Joseph', 'Weizenbaum', date '1923-01-08', NULL);
```

FIRSTNAME	LASTNAME	BIRTHDAY	OBIT
Alan	Turing	1912-06-23	1954-06-07
Konrad	Zuse	1910-06-22	1995-12-18
John	von Neumann	1903-12-28	1957-02-08
Wilhelm	Schickard	1592-04-22	1635-10-23
Ada	Lovelace	1815-12-10	1852-11-27

Figure 8.10 The example table **Persons**

FIRSTNAME	LASTNAME	BIRTHDAY	OBIT
Alan	Turing	1912-06-23	1954-06-07
Joseph	Weizenbaum	1923-01-08	?
John	von Neumann	1903-12-28	1957-02-08

Figure 8.11 The result of the **Persons** query with birthday after "January 01 st, 1900"

It is not possible to tell the system to place the tuple at a certain position in the table. In fact the ordering is not even defined as the relation is a set.

Likewise we can update or delete tuples from the relation. Some examples are given below:

```
update Persons
set obit = date '1995-12-18'
where firstname = 'Konrad' and lastname = 'Zuse';

delete from Persons
where obit = date '1995-12-18';
```

If we have more than one tuple satisfying the *where*-clause all those tuples are affected by the "update" or "delete". It is up to the software system how to find the tuples in the relation, we only need to specify the condition that must be met.

An even more convincing example for the declarative approach is the query. Lets ask for all persons with birthday after January 1st, 1900:

```
select * from Persons
where birthday > date '1900-01-01'
order by lastname ascending;
```

The reason for the compactness of the query lies in the declarative approach. Specifying **how** to find all entries that fulfill the birthday criteria and than writing an algorithm to sort the result would be much more elaborate program than the three lines of SQL query.

The query will return all tuples in the desired birthday range. The result should include the newly inserted Joseph Weizenbaum but not the just deleted Konrad Zuse. The order of the output will be the last name in ascending order as shown in Figure 8.11. We get *Alan Turing* in the first line but *von Neumann* is after *Weizenbaum* because the small "v" has a larger numerical value than the capital "W" in ASCII coding (see Figure 9.1). To get the usual lexical sort order we should change the default collation sequence.

8.4 Regular Expressions

Programming languages are formal languages for writing computer programs. A *formal language* is precisely defined by a set of "words" over an "alphabet". An alphabet is a finite set of symbols like the latin alphabet with the symbols $A, B, \cdots, Z$ or the arabic numbers with digits $0, 1, 2, \cdots, 9$. Word are sequences of symbols from the alphabet that conform to some grammatical rules. An important class of rules are defined by so called "regular expressions".

We will explain the formal rules by example. Let B_T be an alphabet of symbols $\{0, 1\}$. These symbols are called *terminal symbols* as they stand for itself. In our example we can interpret these symbols as two digits 0 and 1. We form "words" by concatenation of the symbols, e.g. 101. The set of all words over B_T contains any sequence of possible binary digits (dual numbers and numbers with leading 0s). If we add the empty word {} we get the set $B^* = \{\{\}, 0, 1, 01, 10, 11, 001, 011, 100, \cdots\}$. We call any subset of B^* a *formal language*. In particular, the set of all *dual numbers* $\{0, 1, 10, 11, 100, 101, \cdots\}$ represent a formal language.

The dots in our example indicate the missing words. Our example has an unlimited number of words. So, it is not possible to write all down. To specify exactly all possible words a formal syntax with recursive description capabilities is needed.

The Extended Backus-Naur-Form (EBNF) is a way to define such sets. It consists of the following syntax:

"literals"	literals are written in quotes
$A := B$	the left side A is defined by right side B
$A \mid B$	separates alternatives
AB	defines a concatenation of expression A and B
$[\cdots]$	defines options
$\{\cdots\}$	defines repetition
$(\cdots)$	defines a grouping
.	defines the end of the rule

As example we define the signed dual numbers with the help of the EBNF:

digit := "0" | "1".
unsignedDualNumber := digit | unsignedDualNumber digit.
dualNumber := ["+" | "-"] unsignedDualNumber.

How do we have to read this rules. Assume we want to construct a *dualNumber*. The third rule defines that a dualNumber is an *unsignedDualNumber* or a plus sign ("+") followed by an *unsignedDualNumber* or a minus sign ("-") followed by an *unsignedDualNumber*. Now we look for the definition of an *unsignedDualNumber*. The second rule recursively states that it is either a *digit* or an *unsignedDualNumber* followed by a *digit*. The digit itself is defined as the literal "0" or "1". Now, that we have these two possible unsigned dual numbers, we use it to construct more numbers with the second option. Replacing *unsignedDualNumber* with "0" or "1" yields:

unsignedDualNumber := "0" digit | "1" digit

We get 00, 01, 10, and 11. Together with the other numbers we have in total 0, 1, 00, 01, 10, 11. Again we can insert the known unsigned dual numbers in the second rule and so on.

Our definition allows numbers with leading zeros, e.g 00101. As exercise of your own change the definition to exclude such anomalies.

A *regular expression* is an expression to describe formal language sets. It is similar to the EBNF but has the following, more powerful syntax:

$A \cup B$	Is the union of sets A and B.
$A := B$	the left side A is defined by right side B
$A*$	is the powerset of A
$a*$	is the powerset of the set $\{a\}$
a^n	is aa^{n-1}, $a^0 := \epsilon$ (empty sequence)
$[\cdots]$	defines options
$\{\cdots\}$	defines repetition
$(\cdots)$	defines a grouping
.	defines the end of the rule

The powerset is the set of all subsets. As example take $S := \{a, b\}$, then the powerset S* := $\{\emptyset, \{a\}, \{b\}, S\}$.

Now we can define again the set of all signed dual numbers using regular expressions:

digit := “0” ∪ “1” .
dualNumber := (“+” ∪ “-” ∪ ϵ) digit digit* .

Please note that regular expressions lead to more compact definitions than the EBNF format. Combining both definitions yields a even more compact result:

dualNumber := [“+” ∪ “-”] (“0” ∪ “1”) (“0” ∪ “1”)* .

A last example that demonstrates the power of regular expression is the definition of dual numbers with an even number of zeros.

evenZeros := 1* 0 (1* 0 1* 0)* 1* 0 1*

Please remember that 1* is an arbitrary sequence of 1’s including the empty sequence.

Figure 8.12 shows that the number 000101101110 is a member of the set of *evenZeros*. To proof this assertion we take the number and replace parts of it with regular expressions until we get the definition of *evenZeros*

The power and expressiveness of regular expressions result from recursive expressions and the use of powersets.

8.5 Spreadsheets

Computers have been very useful to run elaborate programs that solve complex calculations. But until 1979 the user could not accomplish simple daily calculation like household accounting or travel expenses without a dedicated

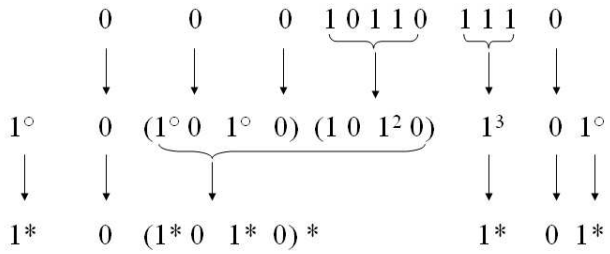


Figure 8.12 The test number 000101101110 is element of *evenZeros*

	A	B	C	D	E	F	G
1	**Data**	**Subject**	**Receipts**	**Accomodation**	**Living**	**Studies**	**Car**
2	1-Jan-05	grant	$500,00				
3	1-Jan-05	rent		$400,00			
4	2-Jan-05	study fee				$2.000,00	
5	3-Jan-05	food			$30,00		
6	8-Jan-05	books				$140,00	
7	12-Jan-05	gas					$40,00
8	15-Jan-05	restaurant			$30,00		
9	15-Jan-05	car insurance					$140,00
10	15-Jan-05	support (parents)	$1.000,00				
11	16-Jan-05	electricity		$50,00			
12	20-Jan-05	telephone, internet		$40,00			
13							
14	**January 2005**	**Totals**	**$1.500,00**	**$490,00**	**$60,00**	**$2.140,00**	**$180,00**
15		% of receipts		33%	4%	143%	12%
16							
17		Balance	-$1.370,00				
18							

Figure 8.13 Spreadsheet layout

program. It was not feasible that professional programmers created a program solution for every little task.

Spreadsheets can help to fill this gap. A *spreadsheet* is a software for tabular information that can be interactively entered and processed. In modern spreadsheets the presentation of the data is not limited to tabular data but can be visualized in virtually any kind of graphics.

Figure 8.13 shows an example spreadsheet layout with letters as column names and integer numbers as row names. Each cell can be addressed by column letter and row number. So technically the name of a cell is its column letter and its row number. The user enters either data or a formula into a cell. The data value can be text or number.

The spreadsheet software usually provides a set of predefined functions to relieve the user largely from defining a formula by hand. Instead the user chooses from a set of functions. Some products provide hundreds of functions for financial mathematics, geometry, text manipulation, logical functions, and others. To distinguish a formula from data the formula is preceded by an equal sign (=). If such a formula is entered the result of the function is displayed in the same cell.

The real power of a spreadsheet comes from cell references. A formula for instance may reference data values of other cells in the table by naming the row and column. That is, a formula may have cell references as parameters. If the cell value changes, the parameter value changes resulting in a new input values for the formula.

An address range is denoted by the first and last address, separated by a colon, like in the following trend function. The linear trend function

=TREND(B2:B13;A2:A13;A15:A19)

computes for the x-value range A15:A19 the corresponding y-values. The trend source are the y-values B2:B13 with corresponding x-values A2:A13.

A powerful mechanism is the relative addressing scheme that helps to copy a formula to an other cell and moving the input parameter cells relatively to the formula's address. This is very useful when the same formula is applied for a sequence of values. This mechanism is comparable with the iteration over a list of values like in a for-loop.

Assume we like to keep the household accounts where we enter all expenses and receipts. We enter a line for each booking and assigning it to a column according to its category similar to the example in Figure 8.13.

The top line of the figure contains text that serves as titles for the columns. We can add totals for the columns and calculate the percentage for each category column with regard to the total receipts. Some cells are for data input while others show calculated results. As example let's take the cell $D15$. We want to show the percentage of expenses compared to the receipts for the accommodation category. So we enter the formula:

= D14/C14

The result will be the value $0.33 = (490/1500)$ in our example. If we format the cell as percentage the result is shown as desired. Now we can copy the formula to the other columns, and alas, the result is not exactly as desired. But how could that be? If we check the formula we notice that it was not only copied to the right, but also the addresses for the referenced cells have been adjusted. This is correct for the numerator of our quotient but unfortunately the denominator has moved too.

The problem lies in the type of reference. We have to distinguish between *relative* and *absolute* addressing. When a formula with a relative address is copied this address moves relative to the destination of the formula. In our example only the numerator should be adjusted and the denominator should remain fixed. If we want to use an absolute address for cell $C14$ we have to write $\$C14$. The revised formula in position $D15$ is then:

= D14/$C14

If we now copy the formula to cell $E15$ everything works fine. The numerator moves relatively and the denominator remains fixed (i.e. still points to cell $C14$) and the resulting formula is:

= $E14/$C14

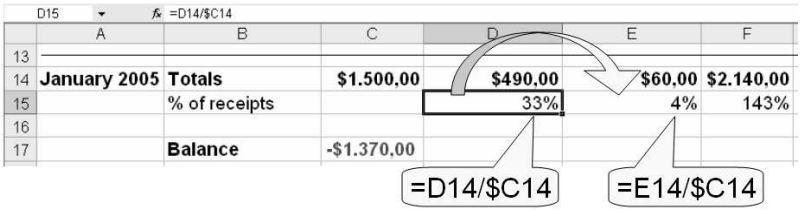


Figure 8.14 Relative and absolute address copying

The concept of copying a formula with absolute and relative addresses involved shows Figure 8.14.

The main benefit of the spreadsheet concept is that any change in value or formula starts a recalculation and the changes are shown immediately. The user can "play" with the data and formulae to test, simulate or do "what-if" analysis.

8.6 History

In the short history of computing probably thousands of programming languages have been developed and still new ones appear. What is the reason for that?

First, only about 20 languages or so are of practical relevance. And in chapter 11 we will see that all are universal languages in the sense that any task that has an algorithmic solution is programmable with any of these languages. But why are always new languages created?

The goal for inventing new programming languages is still to have a better machine independence and a more powerful expressiveness. Rising the abstractness and expressiveness of a language results in more compact and understandable programs. Some application domains are so important that it is economical to develop languages with domain specific elements. One further reason was the way how to develop programs more interactively and the different programming paradigms that need their own language constructs.

Examples for domain specific languages are COBOL (COmmon Business Oriented Language) and FORTRAN (FORmula TRANslating system). COBOL is a verbose language and supports financial calculations and table processing in commercial applications. FORTRAN, the oldest language for technical and scientific calculation, supports multidimensional arrays, complex numbers, and trigonometric functions.

The attempt to develop a multipurpose language produced "monster" languages like PL1 (Programming Language 1) and Algol-68 (ALGOrithmic Language) which were too bulky to become widely used. As a reaction to these language Niklaus Wirth from the ETH Zürich designed the well structured but still lean language Pascal. It served not only at universities as first programming language but found its admirer all over the world. Pascal's successor *Modula-2*, enhanced with information hiding features, displaced Pascal widely but is now superseded by newer object-oriented languages with strong support for Web-programming and Web-services. These languages are Java, developed by

James Goshling (Sun Microsystems), and C# developed by Microsoft Corp. as part of their .NET platform. C# has roots with Java and C++.

The roots of object oriented programming goes back the the 60^{th} when Simula and Sketchpad where developed. Twenty years later the time was ready for this programming paradigm. Smalltalk-80 developed at the PARC (Palo Alto Research Center) by Alan Kay and his team was the most revolutionary creation as it is not only a language but a interactive system. Compared to the other contemporary languages like Modula-2 (N. Wirth, ETH Zurich), C++ (B. Stroustrup, Bell Labs) Smalltalk is radically object-oriented. Everything is an object, even numbers, classes, and code pieces (blocks). Java and C# benefited a lot from Smalltalk even if they could not achieve the elegance and purity of its predecessor.

The C language line started in the early 70^{th} when K. Thompson and D. Ritchie develop it for the Unix operating system. From its structure it is since long outdated but its relation to Unix and Linux boosted its dissemination that it is still used despite the rise of its object-oriented successor C++.

Bibliographical Notes

Nicklaus Wirth designed the programming languages **Pascal** and **Modula**. He published Pascal in the year 1971 in *Acta Informatica* Wirth . His book *Programming in Modula-2* is the official documentation of the enhanced Modula Wirth [1982].

If you are interested in the roots of object-orientation then read the web-page of Dahl about *the Birth of Object Orientation* or look for a copy of the original paper Dahl and Nygaard [1966] about **Simula**.

The implementation and the syntax of the pure object-oriented language Smalltalk-80 was published by [Goldberg and Robson]. Stroustrup enhanced the programming language **C** with object features and called it **C++**.