Computer Programming

using GNU Smalltalk

Canol Gökel

Last updated: 18.10.2009

Cover photo by: Tibor Fazakas over www.sxc.hu

To the ant I accidentally crushed yesterday...

Canol Gökel

Preface

Computers are machines composed of hardware and software written for the hardware. Whether you are an amateur user who uses his/her computer just to surf the Internet or an average user who uses computer for daily tasks, you may need to write a program for a specific duty. Maybe you are just a curious user who wants to dominate the box in front of him before it starts dominating you. Programming is the key word here because you make the computer do what you want by programming it. Programming is like putting the soul inside a body.

This book intends to teach you the basics of programming using GNU Smalltalk programming language. GNU Smalltalk is an implementation of the Smalltalk-80 programming language and Smalltalk family is in general a little bit different than other common programming languages like C or Java. So, we will apply a different approach to teach you programming compared to other computer programming books out there.

You will see chapters of this book are mostly short compared to other programming books. This is because of mainly two reasons. First one is my laziness. Second one is that Smalltalk is a small and orthogonal language. By small, we mean there are fewer programming concepts you should learn compared to other languages. Smalltalk is built on a few carefully designed rules and concepts. By orthogonality, we mean there are very few exceptions on Smalltalk compared to other languages. Because of this two reasons you will learn almost the whole language in less than 100 pages.

This doesn't mean the things you can do with Smalltalk is limited. In contrast, this small set of rules and orthogonality gives you great flexibility so that the only limit is your imagination. Also, one of the greatest strength of Smalltalk is its powerful and rich library which gives you most of the tools you need out-of-the-box. GNU Smalltalk adds even more tools to this valuable toolbox. But because we will concentrate in the core language in this first edition of the book we are going to show you only the tip of the iceberg, namely, only the most important and most frequently used functionality of the library.

If you are an experienced programmer who wants to learn Smalltalk, then you will be surprised by the elegance of this carefully implemented language. Andrew S. Tanenbaum says: "Fight Features. ...the only way to make software secure, reliable, and fast is to make it small.". Smalltalk is certainly designed by scientists with this mentality in mind.

Who is This Book for?

This book assumes that the reader does not have any experience with programming but does have experience with using the computer, meaning, how to open/close it, how to run programs, how to save files etc.

For experienced programmers:

We will often use this kind of box to speak to experienced programmers, referring a person who has knowledge of another programming language than Smalltalk. Newcomers do not have to read the contents of this box.

For experienced programmers:

This book can also be used by experienced programmers. Actually, this book is the one you should read because, most probably, GNU Smalltalk is pretty different than what you have seen so far and this book teaches you Smalltalk as if it is your first programming language.

How to Use This Book

If you are an absolute beginner then we suggest you to read the chapters in order and look at the appendixes anytime they are referred. We tried to write this book in an order that you won't need to look at a future chapter because of unseen concepts, so you may need to turn a lot of pages to look at appendixes but not for anything in the main text. We tried to keep the examples as simple as possible, so that they won't confuse you with unnecessary details and just demonstrate the concept in discussion.

During the learning process of a programming language, it is very important to practice the theoretical subjects you learned. We tried to give a lot of examples after introducing new concepts. Trying to code and run this sample programs by yourself will not only assure that you understand the concept, it will give you both confidence and motivation needed to continue. There are also little questions waiting for you inside the chapters to guess (and probably apply) its solution. Finally, we have prepared review questions so that after reading a chapter, you can intensify your knowledge about the new concepts. We strongly suggest you trying to solve this questions. The answers of the review questions can be found in Appendix B at the end of this book.

Also, most of the programs are available online at:

http://www.canol.info/books/computer_programming_using_gnu_smalltalk/source _code.zip

Font Conventions

We used some font conventions throughout the text so that you can differentiate different kind of materials we are talking about, easily.

Beside the Times New Roman font we use for normal content, we used an *italic* text to emphasize a first appearance, or a definition of a word.

We used a `fixed sized font` while mentioning about a code piece.

We used an *`italic fixed sized font`* while mentioning about a code part you are supposed to put a different code according to your tastes, the context of the program or your computer's settings etc.

The codes which are meaningful even as they are, are showed with a sweet purple rectangle at their left side, like below:

```
a stand-alone meaningful code
```

Outputs (results) of complete programs are given with a purple background, like this:

```
Output of a program
```

The input part of a program which is entered by user while the program is running are showed in bold face at the output:

```
Please enter your name: Canol
```

When we mention about keys on your keyboard, we'll write them between angle brackets like <CTRL>, <Enter> or <Backspace>[1].

We also have some special boxes:

Note:

This kind of boxes contain important details, some additional information or suggestions.

For experienced programmers:

1 We will use <Enter> throughout the book for both <Enter> key on PCs and the <Return> key on Macintosh computers.

This kind of boxes contain some hints for people who knows some other language but reading this book to learn GNU Smalltalk. Beginner users or even experienced programmers don't have to read or understand things written in this boxes.

Question:

This kind of boxes contain some special questions which are answered at the and of its chapter.

Contents

The Star Trek computer doesn't seem that interesting. They ask it random questions, it thinks for a while. I think we can do better than that.

Larry Page

Chapter 1: Introduction to Programming World

In this chapter we will dive into programming world. See what programming is, what programming languages are, talk about some basic topics you should be familiar with before starting programming.

What is Programming and What are Programming Languages?

Programming is making the computer do whatever we want it to do. And *program* (or *software*) is the name given to a series of computer code to complete a specific task. Sometimes the procedure of writing a program is also named as solving a problem because all programs are written to do a specific task, in other words to solve a problem you faced in real world.

Because of the hardware design, computers can only understand two states. We represent this two states with digits 0 and 1 (We will mention about this, more detailed, later in this chapter.). Combinations of zeros and ones generate some commands that computer hardware understands. We call these command groups which provide a communication between us and computer hardware as *programming language*s.

We can write commands with just zeros and ones but this is a very painful method for humans. Imagine a stack full of zeros and ones. To be able to recognize the commands from it at first sight would took years of experience for a reasonable human being. So, humans solved this problem by developing an intermediate step between writing programs and executing it. They write it in a more understandable, most probably in an English like grammar, and then convert it into hardware understandable zeros and ones, which is called machine language or machine code. This converting process is done also with programs called compilers or interpreters.

Programming Language Types

Programming languages are classified according to their various properties. We will just mention a few of them.

High Level and Low Level Programming Languages

As we mentioned earlier, machine codes are hard for humans to understand. So, humans developed languages which are like languages that we today speak. A language can be categorized according to how close it is to a human language. As closer a language is to a machine language, as lower level it is. In other words; as closer a language is to human languages, as higher level it is.

Below you see what a machine code looks like:

```
0110101110100010110000101
```

You cannot even guess what this means right? Now, below we will give an example how the language called *Assembly* looks like:

```
MOVE  d'3', W
ADD   d'4'
```

I don't know if you can guess what this code does but it is obvious that it is more readable than a machine code. So we can say that Assembly language is a higher lever language than machine code. By the way, if you are curious about what the code above does, it just sums up two numbers.

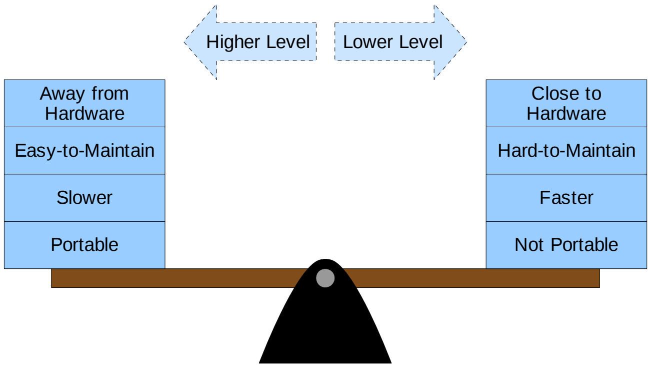
Now look at the same code above written in a language named GNU Smalltalk (Hmm, I never heard of it before):

```
3 + 4
```

I won't ask if you have guessed what the code above does because I don't know of any way to describe it better than the code line above, if you say no. So, the highest level language when we compare machine code, Assembly and GNU Smalltalk is GNU Smalltalk.

Although, the examples we have given to describe the language levels are pretty simple and obvious, don't expect GNU Smalltalk as almost the same with English. It can get sometimes as confusing as Assembly, although it has a reputation as being a very high level language compared to other programming languages commonly used today.

We will finish this topic with an important statement, being a higher level language does not mean being a *better* language. Assembly is also a very commonly used language today because when a language is close to hardware, the control of you over hardware becomes stronger. So, for doing very fundamental hardware programming Assembly is probably a good choice. Also the lower level the language you use the higher control of you over optimization of the code which results in faster programs. Of course, using Assembly to code an office suit like Microsoft Office just because of the advantages of Assembly would be a silly choice because it will result hundreds of times longer, more complex, unreadable and that's why unmaintainable code when it would be coded in a higher level language. So, you should consider all the pros and cons before choosing a programming language for the current project. This also means that you may need

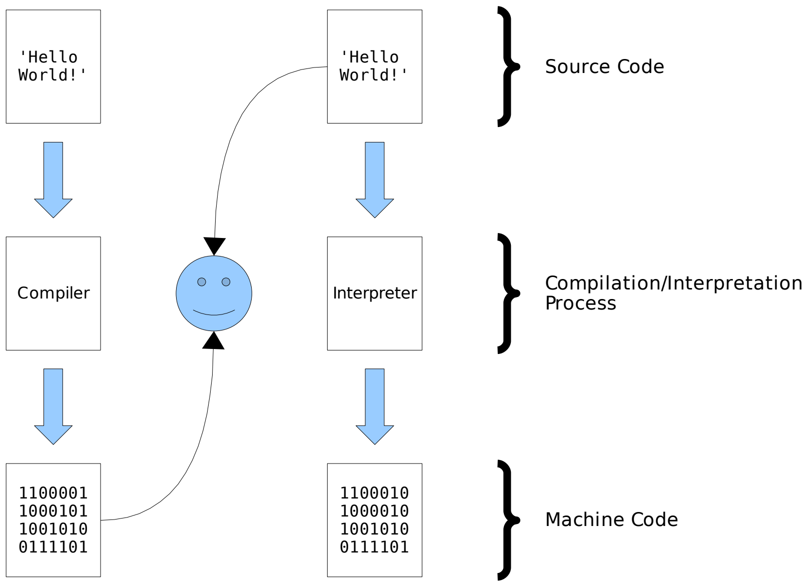


Compiled and Interpreted Languages

We mentioned that writing a program is done by using programming languages and there is a step of converting the programs we write from human understandable form to computer understandable form. This process is called *compilation* if the entire program is converted into machine language at once and kept in computer memory in that form. If the software written in a programming language requires to be compiled before run then this language is called a *compiled language*. The software doing the compiling process is called *compiler*.

But if the program is converted as we write, that is line by line or more correctly statement by statement then this process is named as *interpretation* and such languages are called *interpreted language*s. Software written in interpreted languages are kept in computer memory as the way we write them and every time we execute (or launch, run) them, they are interpreted again into machine language. The software doing the interpretation process is called *interpreter*.

The code we wrote in a programming language (not the result of a compilation or interpretation process) is called *source code*.



The figure above demonstrates the difference between launching a compiled program and an interpreted program. A user can reach directly to the compiled program which is kept as machine code in computer memory while she can reach only the source code of an interpreted language. The system then recognizes the file format and sends it to the interpreter to interpret it into machine code. Sometimes, the system may not be able to recognize the file format and we might need to send the file to the interpreter manually. The main thing to notice is that interpreted languages require additional steps to be launched and it slows down the execution time.

Actually, there could be both compilers and interpreters for the same language depending on its structure but most languages are designed with one of them in mind, so there could be differences between the compiled version and the interpreted version of a language.

Although the two types of programming languages are much popular in this scene, there is also another type which is between this two paradigms. This paradigm compiles the source code into an intermediate level which is still not understandable by hardware nor by humans. This level of conversion is generally called *byte-code* compilation. This process is done so that the code's level is made more close to hardware and converting it to completely understandable form by hardware is much faster. The converting process from byte-code to machine code is done by programs called *virtual machine*s. The thing virtual machines do is interpreting the byte-code into machine code when the program is executed by user.

Why there is this third type of language? Because interpretation is slow compared to executing a compiled program but has an advantage of being cross-platform. *Cross-platform* is the name for being able to run a software on different computer architectures, like different operating systems or different processors. So you can launch a program written in an interpreted language if there is an interpreter for it on the computer you are working on. The hardware you have or the operating system you use does not matter. If you had programmed the software with a compiled language you should have done compilation first for the processor and operating system you have before launching the program. This has its own advantages like being executed much faster after compilation and disadvantages like compiling the program every time you make a change in source code and every time you move your program from one platform to another. Compiling a program into byte-code speeds up this interpretation process and still has the advantages of an interpreted language.

Although, GNU Smalltalk functions more like an interpreter, Smalltalk is designed as a language which uses a virtual machine. We can give Java programming language as an example of this type of languages (and a very successful one, indeed).

Procedural, Functional and Object Oriented Programming

Another categorization of programming languages are according to their paradigms. A *programming paradigm* is how a programming language looks at the problems to be solved. There are mainly 3 types of programming paradigms: Procedural, functional and object oriented. To keep it simple and not confuse you with words you don't know yet we will only mention about object-oriented programming which is the heart of Smalltalk language.

Object oriented programming looks at the world as an object compound of other objects. According to this paradigm everything can be considered as an object. For example, a computer, a television, a book etc. are all objects. Also all of this objects are compound of other objects. For example, a computer is made by bringing a main board, a graphics card, a hard disk and some other hardware together, which are all again objects.

Object oriented programming is generally considered as the most close programming paradigm to human thinking. So, when you write object oriented programs you feel more comfortable and you can focus into solving the problem instead of worrying about the programming language structure, rules etc..

There is no rule that a programming language should be based on only one paradigm and built itself on it. Actually, there is even a name given languages which allow the coder to use more than one paradigm during programming, *multi-paradigm programming language*s. Programming languages which allow programmer to use only one programming paradigm are called *single-paradigm programming language*.

Smalltalk is one of the first object oriented programming languages and it is a purely object oriented programming language. So, it is a single-paradigm programming language and you will look everything as an object.

Number Systems

We briefly mentioned that computer hardware can only understand ones and zeros. Now we will explain the reason a little more detailed. The hardware used commonly in computers today can track only two states because of their design. We can call this states as on and off, for example. So humans needed a way to describe the world in just two symbols. Ways to describe mathematics in two symbols are already implemented so they just needed to represent other things in two symbols. For example, how can you represent a refrigerator in computer world? We can represent it as a collection of some strings. Then how can we represent strings with just two symbols? Humans developed ways to represent strings in two symbols which we will show you in topic *File Formats*. We presented some *states* but we should be able to change these states. We achieve this via defining some basic *commands* and constructing more complex ones on top of the more basic ones. This is the way from machine language to Assembly language and to higher level languages like the ones we mentioned before in this chapter.

Question:

Although we tried to give you an idea of what happens in a computer, we won't cover this techniques in detail. Maybe you can try yourself to imagine how to create a world in just two symbols. This will be really helpful to create an understanding of computer and will be an interesting practice for you. After imagining yourself a little bit you may look at the Assembly language to see how Assembly solved this problem. Of course, it will be very hard for beginners so we recommend that you finish this text before.

Now we will mention about the *arithmetic* part of the two symbol world. Though we won't cover it detailed not to bore you with mathematics concepts. You can always look at any algebra book to get a deep insight of the materials we will introduce you next.

Base-10 (Decimal) Number System

We use ten symbols (0, 1, 2, 3, 4, 5, 6, 7, 8 and 9) to represent the numbers in our daily life. Why ten and not nine or eleven? Is this how the world is created, in ten symbols to represent numbers? Of course not. We can represent numbers in as many symbols as we want. The reason we use ten symbols is that we are used to it. It is thought to be used by first Indians and spread the world from there. Why Indians choose ten is thought to be because humans have ten fingers.

Base-2 (Binary) Number System

Mathematics allows us to use as many symbols as we want to represent numbers. Because we can use just two symbols in computer architecture, base-2 system is ideal for this job. Base-2 numbers are usually called *binary number*s. We choose usually 0 and 1 as the two symbols but have chosen whatever symbol we want.

We want to show you first how to convert the value of a number written in base-*x* (*x* being a natural number) to base-10. When we write a number in base-*x*, you can calculate the value of that number in base-10 by multiplying the digit value of each digit by the base number raised to digit's algebraic order and summing all of them. Definition is a little bit confusing, isn't it? Let's give an example. Suppose we have a number written in base-10 like this: 345. We can evaluate the same value of this number like this:

$$345=5\cdot10^0+4\cdot10^1+3\cdot10^2$$

So, in base-2 we can write the same number like, 101011001:

$$101011001=1\cdot2^0+0\cdot2^1+0\cdot2^2+1\cdot2^3+1\cdot2^4+0\cdot2^5+1\cdot2^6+0\cdot2^7+1\cdot2^8$$

This is how we convert a base-x number into base-10. Now, we will show you how we can convert a base-10 number into a base-2 number. Suppose that we want to represent number 345 in base-2. All we have to do is to divide 345 to 2 repetitively until we reach a number that can be represented in base-2. Then we write the resulting number and the remainders of previous divisions next to each other from right to left. To show this process:

$$345/2=172\ \ (\textit{remainder}\ \mathbf{1})$$

$$172/2=86\ \ (\textit{remainder}\ \mathbf{0})$$

$$86/2=43\ \ (\textit{remainder}\ \mathbf{0})$$

$$43/2=21\ \ (\textit{remainder}\ \mathbf{1})$$

$$21/2=10\ \ (\textit{remainder}\ \mathbf{1})$$

$$10/2=5 \quad (remainder\,\mathbf{0})$$

$$5/2=2 \quad (remainder\,\mathbf{1})$$

$$2/2=\mathbf{1} \quad (remainder\,\mathbf{0})$$

When we write the result of last division and the remainders of the divisions back to beginning from right to left (we boldfaced that numbers in the above equations) we get 101011001, which is the same as the 2-base equal we proposed.

Each digit of a binary number is called a *bit* and 8 bits next to each other are called a *byte* in computer jargon.

Base-8 (Octal) Number System

Sometimes we use numbers written in base-8 to represent binary numbers because octal numbers are shorter and there is an easy converting method from octal to binary number system and vice versa. There are also some historical reasons for that, so, learning it would be, sure, an honor for us.

We use digits 0, 1, 2, 3, 4, 5, 6 and 7 to represent octal numbers. We can convert a number written in binary to octal by grouping every 3 digits beginning from right of the number to the left and replacing every group with the octal equivalent of it. For example, the number 101011001 written in binary system may be represented in octal system like:

$$\underbrace{101}_{5}\underbrace{011}_{3}\underbrace{001}_{1}=531$$

If the number cannot be evenly divided into groups of three digits we can add 0 to the left of number until it can.

The reverse operation, converting an octal number to a binary number is also straightforward. You just convert every digit of an octal number into its binary equivalent and write them next to each other. For example, the converting process of octal number 531 to binary can be showed like this:

$$\underbrace{5}_{101}\underbrace{3}_{011}\underbrace{1}_{001}=101011001$$

Question:

Can you explain why such conversions are always true?

Base-16 (Hexadecimal) Number System

Now, we will see a number system which is usually harder to imagine, at first. Base-16 means we will use 16 symbols for writing numbers. But we have only 10 of Arabic number symbols. So what are we going to do? It is acceptable to use any symbol we want for the rest of it. Usually, we use the first 6 letters of Latin alphabet; A, B, C, D, E and F for values 10, 11, 12, 13, 14 and 15, respectively.

In this case, while you say that there are "12 chocolates in front of you", maybe an alien who comes from a planet using base-16 numbers in their daily life would say that there are "C chocolates" in front of it and it would be totally acceptable.

Converting from base-2 to base-16 and vice versa is very alike with conversions between base-2 and base-8 except that this time you deal with 4 digit groups.

File Formats

A *file format* is how a file is kept in the memory of a computer. Files may be categorized mainly in two: Binary files and text files. We will use files extensively during our programming practice. Actually, you are using files every time you use computer because every program you use are composed of a file or some files. So it might be handy to know a little bit more about the files.

Binary Files

*Binary file*s are files which are used by some special programs written to read that special file format. They are represented as zeros and ones in computer memory and their content is created according to some specifications. .jpeg, .pdf, .doc, .exe are all examples of binary files and need special programs like image viewer or PDF reader to be used.

Text Files

Humans use some files which are just composed of characters so extensively, that they created simple text files. Text files include just alphanumerical characters and some special control characters which are all presented according to a special encoding system which are called *character encoding*. The characters they can include are limited with the capabilities of the character encoding they use. A list of characters which can be presented are listed in tables called *character set*s. Every character is presented with a number in a character set. There are special programs called *text editor*s which are capable of reading and writing text files.

The most commonly used character encoding is ASCII (American Standard Code for Information Interchange) which uses a simple encoding that can be expressed in a single sentence: Every character is presented in seven bits. A text editor which supports ASCII character encoding can read any ASCII encoded text file. This along with the simplicity of the character encoding specifications provide an enormous portability between digital equipment. You can find the ASCII table at Appendix B of this text.

Because ASCII has a 7-bit limit and you can only write 128 different number with 7 bits, ASCII can represent only up to 128 different characters. The world is big and there are hundreds of different alphabets with thousands of different characters. There are some languages which use alphabets that have more than 128 characters so it might be even impossible to represent a single alphabet in a 7-bit encoding. That's why there are a lot of different character sets and encodings. For example, UTF-8 character encoding, which may be used along with Unicode character set, can represent characters via 1 to 4 bytes. Unicode character set has a character collection of about 100.000 characters.

Our GNU Smalltalk interpreter can read text files encoded with ASCII or UTF-8 so it would not be a problem whether your editor supports ASCII or UTF-8.

Word Processors, Text Editors and IDEs

We will type our source codes into text files and we are going to use a program to write our source codes just like we use a program to create documents.

When typing is in discussion most probably a *word processor* like Microsoft Word, iWork Pages or OpenOffice.org Writer comes to an average computer user's mind. But word processors are not created for simple text files. They use binary files to keep the fancy document formatting like bold, big, colored text, images and the layout of this kind of things. They most probably have an option for creating plain text files but then we can't take advantage of the usefulness and speed of text editors. This is like using a limousine to harvest.

*Text editor*s provide an easy to use and fast solution for a programmer. There are also some text editors which are especially designed for programmers. They have some useful features to make a programmer's life easier.

So we will use text editors to create and manipulate our text files. Most of them are specialized for different character sets and encoding. On Linux, you can use Gedit on Gnome, Mousepad on Xfce or Kate on KDE. There are also some advanced text editors like Vim or Emacs. On Windows you can use Notepad but Notepad is awfully less advanced so you can download a little more advanced ones like Notepad2[2] or EditPad Lite[3].

Using a search engine like Google or Yahoo with keywords "text editor" will always give you good results.

A programmer usually needs some others programs during coding, compiling and

2 http://www.flos-freeware.ch/notepad2.html

3 http://www.editpadpro.com/editpadlite.html

testing of programs. Instead of installing and using separate programs you can use programs called integrated development environment (IDE) which include almost all the necessary things for you. Some of the well known IDEs are Anjuta or KDevelop for Linux, Microsoft Visual Studio for Windows or cross platform ones like Eclipse and Komodo. We won't use such environments because it is not necessary when you are just at the beginning of learning how to program.

In the next chapter we will be introduced to some fundamental concepts of GNU Smalltalk programming language and write our first program.

Review Questions

1. What is a programming language, why do we need it?
2. Explain briefly the differences between compiled and interpreted programming languages. What are the advantages and disadvantages of them? How can we classify GNU Smalltalk in this manner?
3. What are programming paradigms?
4. Can a programming language support more than one programming paradigm? Is the secret feature of GNU Smalltalk, which differentiates it from other languages, supporting more than one paradigm?
5. Write the decimal number 543 in binary, octal and hexadecimal format.
6. Write the binary number 10110100 in decimal, octal and hexadecimal format.
7. Write the hexadecimal number A93F in binary, octal and decimal format.
8. What are binary files and text files? Can you give some other examples of binary files other than given in this chapter? Can you specify a method to determine whether a file format is binary or text?
9. Explain the differences between text editors and word processors. Why shouldn't we use word processors to write programs?

Unfortunately, the current generation of mail programs do not have checkers to see if the sender knows what he is talking about.

Andrew S. Tanenbaum

Chapter 2: Introduction to GNU Smalltalk

We will dive into GNU Smalltalk world and write our first program in this chapter. This chapter is rather short compared to others but it has two reasons. Firstly, you are going to need to read the necessary portion of the Appendix B; secondly, writing your first program is probably one small step for mankind but one giant leap for you.

Small Talk About Smalltalk

We are going to use Smalltalk programming language to learn about computer programming but actually any programming language, although differs in some methodology, can be used as a teaching tool as more or less what we do to program the computer is the same, in the end.

Smalltalk has a very special place in computer science history. It appeared publicly around the early 80s as a product of Xerox PARC (Palo Alto Research Center). It was pretty different than the programming languages that far in regards to its vision in object oriented programming and the syntax and environment it used to realize this method of thinking. Also it was using a virtual machine concept that was not popular at the time.

> Note:
>
> Smalltalk has some features which makes it not only a programming language but a whole programming environment. But to avoid too much repetition of ourselves, we will use the words "Smalltalk programming language" and "Smalltalk programming environment" in exchange.

Smalltalk, interestingly, was the first programming environment to use a graphical user interface (GUI) which is like we all use today: several windows arranged on monitor (which is, as a whole, called desktop) and things like scroll bars, buttons and menus placed in windows. Smalltalk environment was also being used for networking in Xerox PARC which was not also common at the time.

So, when we look at the history of Smalltalk, today, we realize that it was ahead of its time and influenced the programming languages released after. Some of the concepts introduced in Smalltalk is essential today in computer programming.

If you came across some of the words you haven't seen before, in this section, don't worry. We will learn all the concept we need to know as we go. And using this carefully designed, elegant, simple and yet powerful programming language as the teaching tool will help us understand programming concepts easily than any other.

General Logic Behind Smalltalk

Being a purely object oriented programming language, Smalltalk treats everything as an object. The communication between objects are provided by sending messages to them. For example, you create a computer object and tell it (send it a message) to open your office suit and it opens. Maybe you can create a human object before, who owns a computer object. Then you can send *her* a message to send her computer a message to open her office suit. The detail level you create depends on the application needs. Sometimes you may need to create a human sometimes you don't. We will mention more about objects later.

For experienced programmers:

In contrast of other popular programming languages like C++ or Objective-C; Smalltalk does not allow you to do procedural programming.

First Program

Now we are going to write our first program, in 2 ways. But before we start writing, you should make sure you got the programming environment installed. If you have not installed it yet, you should now and Appendix A will help you to do that.

Our first program will make computer display a text to the terminal (or command prompt in Windows terminology). Now open up your terminal and type:

```
gst
```

```
GNU Smalltalk ready

st>
```

GNU Smalltalk interpreter is started and now waiting for commands to execute. This is called *interactive mode*. Now we are ready to write our program. Type the code below and hit <Enter>:

```
'Hello World!' printNl
```

```
'Hello World!'
'Hello World!'
```

Note:

If you are using an old version of GNU Smalltalk, you may need to append an exclamation mark at the end of the code like: this: `'Hello World!' printNl!`.

We strongly suggest that you get a newer version of GNU Smalltalk to be able to run all the examples on this book.

As you can see from the output, the computer printed the string, `'Hello World!'`, two times to the terminal, which might also be named as *standard output* in programming terminology. Now, we will explain this code and why the computer printed the string two times rather than one time.

As we mentioned, everything in Smalltalk is an object and we communicate with them by sending messages. Then, they response us accordingly. In the code above we have a string object and a message. Characters written between single quotes create string objects. So, here, `'Hello World!'` is a string object. We want this object to print itself to the terminal so we should send a message to it. The message for this purpose is `printNl`. The "`print`" part is obvious, the "`Nl`" part indicates that it is going to insert a new line character afterwards.

If you understand so far, you should still be wondering why it displayed the string two times and not just one time. It printed the string for the first time because we want it to and it printed that string second time because the overall expression evaluated the value `'Hello World!'` and, in interactive mode, the last evaluated expression is printed to the terminal. If we would have written a value directly, then it would be printed, as well, because a value always evaluates to itself. We can try it by writing a digit as a command. Actually, a digit is also an object and it will obviously evaluates its own value. Let's try it, write `3` to the command line:

```
3
```

```
3
```

You see? I hope you saw that you can count on me...

That's it. We've already written two computer programs! But we said that we are going to write the program in two ways. So, lets do it in an alternative way. Now, we are going to write the program into a text file and have the GNU Smalltalk interpreter read the instructions from it and then execute it. So, we won't be in interactive mode this time. These text files are called source code. Now, open your favorite text editor like Gedit, Kate on Linux or Notepad on Windows and write the below code in a new file:

```
"hello_world.st"
"A program to print 'Hello World!' to the terminal."

'Hello World!' printNl
```

Now, save it as `hello_world.st`. This is the source code of our program. The characters between double quotes are *comments*. Comments have two purposes; one of them is to make the source code clear to reader so that he/she understands it faster. So, they are omitted by the interpreter. The second function of them is for documentation purposes. We will explain when it is counted as a comment and when a documentation string later. For now, just know that above strings are comments. When we are writing source codes we will write into first line the -suggested- file name, so that you won't waste time thinking about how to name the file (How thoughtful we are...). We will write into the second line a brief explanation of what the program does so that if you wonder what the program does later on, these strings will remind you of it. Note that these comment strings are optional. You can omit them if you want. But don't forget that writing comments into source code is a good programming habit because sometimes you will need to maintain your programs months after you write them and even you, as the writer of that program, will have difficulty remembering why you have written some of the commands. Also if you are working with some other coders in a big project then you will need comments so that you understand each others code, easily.

So, how are we going to have the interpreter execute it? Now, if GNU Smalltalk (*GST*) program is still running, in other words your terminal is still showing something like this:

```
st>
```

then press <CTRL> + d in Linux or <CTRL> + z and then <Enter> in Windows to quit the program. Now, your terminal should be waiting your commands. We will start the GST program again but this time we will send it the path to our source code as an argument. Please go to the directory where your `hello_world.st` file is located and type the command below:

```
gst hello_world.st
```

```
'Hello World!'
```

After executing the codes written in the source code, the GST program will automatically terminate, leaving you alone with the forsaken terminal... Note that, because we are out of the interactive mode, the string is printed only once.

Writing your source code into a file has many advantages. You can execute it as many times as you want and keep them for later use so that you don't have to retype all of the program again. But using the command line to type a program can be very handy for small tasks or experimental coding, which you will frequently do throughout your learning process.

In the next chapter we will explain the object, message and class concepts.

Review Questions

1. Write a program which displays a diamond like below onto the screen:

   ```
      /\
     /  \
    /    \
    \    /
     \  /
      \/
   ```

   (Hint: Use forward/back slash and space characters.)

2. What can be the advantages and disadvantages of writing a program into a source code file or entering it directly to the terminal, in interactive mode?

3. What are *comment*s for in GNU Smalltalk source files? How can we write them?

4. The power of computers comes from being able to deal with huge amount of data, amazingly fast and without doing a mistake. Can you write a program which displays numbers from 1 to 1000 onto the screen? Would you? Imagine how would it be *easily* possible.

5. We called terminal as our *standard* output. Then there should be more output devices. Can you think of a few examples?

If you don't fail at least 90 percent of the time, you're not aiming high enough.

Alan Kay

Chapter 3: Objects, Messages and Classes: Part I

Objects and Messages

An *object* is a unit which represents a *thing* in the program. It has its states, which are called *instance variable*s, and it has some definitions which defines how to respond to certain messages it receives from outside (usually from us or from other objects) which are called *methods*. We communicate with objects via messages we send them. Then the object responses back if he understands our message. We have certain syntax rules to write messages and to define objects, its instance variables and methods. We did quite make some definitions and most probably you didn't understand them completely. If so, we suggest that you read the above paragraph once more and continue because this concepts are hard to catch if you firstly encounter them. After giving a few examples, these concepts will be more clear to you.

Now we will give you some example object & message expressions. How about some arithmetic expressions first? Write the code below to GNU Smalltalk interpreter and hit <Enter> from your keyboard:

```
3 + 4
```

```
7
```

The spaces are not important, you can place as many spaces as you want or don't place any space at all. You can even put new line characters around + character. Now we are introduced to a new concept called white spaces.

*White space*s are the name given to space, tab and new line characters. They are called white space because we simply can't see them. Usually, white spaces does not have a meaning in programming languages, so, compilers and interpreters omit them if they are not inside a string expression.

Now, we can return to our arithmetic expression. Here, our main object in concern is `3`. The object we send messages to are called *receiver* and `3`, here, is the receiver. `+ 4` characters form the message we send to our receiver, `3`. The message says to `3`: "Sum yourself with the number I sent and return the result.".

Selectors and Arguments

Now we will look into the message we sent above a little more deeply. There are actually two parts of this message. One is a selector, which is + character; and the other is an argument, which is the object `4`. You can get the selector by removing argument parts of the message (We will give an example for that later.).

Selectors form a system to help objects decide how to respond to messages. For example, here + selector is actually defined in object `3` so that `3` knows what to do when it receives this message.

Arguments are the additional data the object should use while evaluating the response. Objects know how to use these additional data while evaluating the response. We will see how objects know all about these when we see how we can define our custom objects. Note that a message don't have to have any arguments. Actually, we will try a message without an argument, next. Here it is:

```
'Hello World!' size
```

```
12
```

Here, we have a string object `'Hello World!'` and a message `size`. This message tells string objects to return their length, in other words, how many characters they have. As you can see, `size` message does not have any arguments because it does not need to, in other words, it has all the information needed, in other words, the answer is in the question, alright it's enough.

Let's look at one last example and then we will mention about message types. We said that computers understand only ones and zeros. So, the number `3` is actually written to the memory of the computer in base-2, like this[4]:

```
00000011
```

We have written it with 8 digits representing 1 byte of memory. Now, given the information above, try typing the following code:

```
3 bitAt: 3 put: 1
```

```
7
```

This expression has an object of number `3` and a message `bitAt: 3 put: 1`. We have the selector: `bitAt:put:` and two arguments: `3` and `1` in this message. Remember how we get the selector name? We get rid of the arguments and put together the rest of the message.

4 Actually, the format the integer is kept in the memory might be a little more complex for optimization purposes but for the example we will give, you can rely on this assumption.

This message tells the integer object `3` to reach 3th bit of itself and replace it with `1`. The bits are numbered from right to left starting from 1. We had `0` at the 3th bit of `3` before. But it is now `1`. So we have:

```
00000111
```

which corresponds to 7 at base ten. After sending the message, the expression is evaluated and printed to the terminal.

Unary, Binary and Keyword Messages

There are 3 types of messages. The difference between them is about the selectors, arguments and the precedence. What is precedence? It will be explained in a few minutes.

*Unary message*s are messages without any argument. It is called unary because the resulting expression involves only one object. An example for an expression with unary message would be:

```
'Hello World!' size
```

As you can see there are no arguments here and the overall expression involves only the string object `'Hello World!'`.

*Binary message*s are messages with an argument. But the other characteristic of binary messages is that their selectors have up to 2 characters which are not alphanumeric. An example for an expression with binary message would be:

```
3 + 4
```

`+` is a selector here and `4` is the argument for it. The expression involves two integer objects of `3` and `4`. `+` is one character long and it is not an alphanumeric character. So it meets the requirements of being called as a binary message.

Finally, *keyword message*s are messages with one or more arguments whose selectors are composed of alphanumeric characters. They usually have a colon at the end of each selector word which indicates that it is demanding an argument. An example for an expression with keyword message would be:

```
3 bitAt: 3 put: 1
```

Our selector here is `bitAt:put:` where our arguments are `3` and `1`. Overall expression involves three objects but it could have been one or two or another number greater than three.

We gave the selector and argument features of message types, but we didn't mention about precedence of them. *Precedence* is a concept used for deciding which one of the messages will be executed first, if there are many messages in one expression. If you cannot imagine an expression with many messages don't worry, here it is:

```
3 + 5 bitAt: 3 put: 1 printNl
```

As you might see, here are three messages: `+`, `bitAt:put:` and `printNl`. Which one should be evaluated first? It depends on precedence rules.

There are some basic rules for message precedence in Smalltalk, they are:

1. Unary messages are evaluated first,
2. Binary messages are evaluated second,
3. Keyword messages are evaluated last,
4. If you encapsulate an expression between parenthesis then it is evaluated first,
5. If the messages have the same precedence, then the messages are executed from left to right, respectively.

Now we will give two examples to illustrate this precedence rules. The first one is the above example we have given:

```
3 + 5 bitAt: 3 put: 1 printNl
```

```
1
12
```

Now, we will investigate why the output is `1` and `12`. When our interpreter first meets with this expression it sees the unary message `printNl` and executes it with the object in front of it which is `1`. So the first output is `1`. Then this expression evaluates `1` itself and returns it. Our expression turns into something like this:

```
3 + 5 bitAt: 3 put: 1
```

We have two messages here. One of them is `+`, which is a unary message and the other one is `bitAt:put:`, which is a keyword message. According to the rules above the expression with binary message will be evaluated first which yields the integer `8`. So, our expression turned into:

```
8 bitAt: 3 put: 1
```

Now we have only 1 message which is a keyword message and it evaluates to integer 12 (You can try to do that operation on paper by yourself.). As the last thing to do, the interpreter prints that object onto terminal.

Our second example will be the expression below:

```
(3 + 5 bitAt: 3 put: 1) printNl
```

```
12
12
```

The only difference here is that we parenthesized all the messages before `printNl`. But the result is a very different one as you can see, because the expression in parenthesis is first evaluated and then sent the message `printNl`.

Another Way to Display Output on Terminal

Before continuing to the next section, we will show you another way to display a text on our terminal, which we will use frequently throughout this book along with `printNl` message.

With `printNl` message, we told to a string object to write itself onto terminal. But think of this process in a reverse way: we could send a message, this time, to the terminal to write a string object's value onto itself. Although the name comes from another concept, standard output has the name `Transcript` in GNU Smalltalk. And the message selector to have it display a text is `show:`. As you can guess from colon character at the end of the selector, it needs an argument which is the string object we want to display. Here is an example; execute the GNU Smalltalk interpreter via `gst` command, type the below code to the interpreter and hit <Enter>:

```
Transcript show: 'Hello World!'
```

```
Hello World!
```

You might have noticed some differences than when we used `printNl` message on a string object. For example, this time there are no single quotes around our output. Also there is no new line after the output. We will show you how to put a newline character at the end of the output when using `Transcript`, in the *Message Cascading* section, just a few minutes later. Lastly, the output is printed only once while it was printed twice every time we used `printNl` message. This is because the overall object returned is the Transcript itself and it does not have a representation as a string.

Message Chaining

If you write more than one message next to an object name, all messages will be sent to the resulting object of the preceding message. This is called *message chaining*. The general syntax for message chaining is like this:

```
objectName message1 message2 message3 ... messageN
```

So, while `message1` is sent directly to `objectName`, `message2` is sent to the resulting object of expression `objectName message1`. Using the same logic we can say that `message3` is sent to the resulting object of expression `objectName message1 message2`..

Here is a real life example of message chaining:

```
'Canol' reverse asUppercase
```

```
'LONAC'
```

We manipulated the string `Canol` multiple times with two messages to reach the resulting string object `'LONAC'`.

Message Cascading

GNU Smalltalk allows us to send multiple messages to an object without writing object's name over and over. This is called *message cascading*. The general syntax for message cascading is as follows:

```
objectName message1; message2; message3; ... ; messageN
```

We put semicolon between the messages we want to send. Beware that this is different than message chaining in which semicolons are absent and every message is sent to the resulting object of preceding message.

Here is an example for message cascading:

```
Transcript show: 'Canol'; cr
```

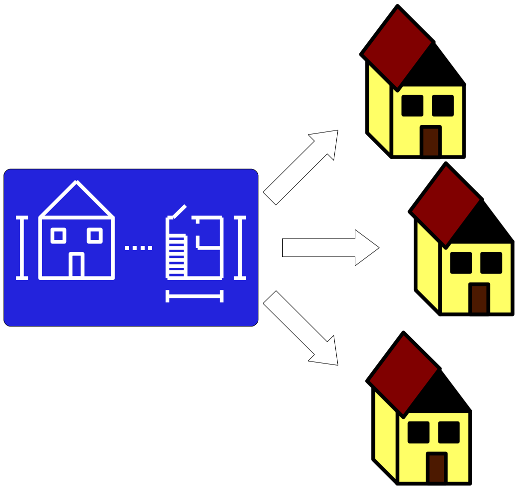
`cr` is a message which causes `Transcript` to put a newline character which is also called *carriage return*. So, we first sent the message `show: 'Canol'` and then the message `cr` to the `Transcript`.

You have to be careful when using message cascading because, sometimes, precedence rules may cause results that you don't expect.

Classes

Classes may be summarized as the *templates* of the objects we want to create. We first define a class, then produce objects from it. That's why objects are also called instances of their classes. It is like the blue print of houses.

Every object has a class related to it. For example, `'Hello World!'` object is created from the `String` class or numbers like `2` or `3` are created from a class called `SmallInteger`. We will mention about the theory of classes later on Chapter 5 in more detail.



We can think of the blue print as a House class and the actual houses as the instances of it.

Common Classes and Their Usage: Part I

GNU Smalltalk has a library which provides a rich set of classes. In this part, we will see some of the most commonly used classes and the messages we can send to their instances. Although there are hundreds of classes and thousands of messages, we cannot cover all of them. We will try to give a lot of examples, though, to make you familiar with GNU Smalltalk code.

Number

Numbers are one of the most commonly used data types in computing. Computers are much faster and more accurate than a human being when it comes to manipulate large numbers and doing mathematical operations on them. GNU Smalltalk has a `Number` class which provides a large set of behavior we can benefit from.

But how are numbers presented in GNU Smalltalk? Here are some presentations of numbers:

We can represent a natural number just like on paper:

```
3

15

1000000
```

We can add a hyphen character to the beginning of the number to make it negative:

```
-3

-15

-1000000
```

We can add a dot character as a decimal point:

```
3.5

-15.7

0.1
```

We can represent numbers in other bases then decimal. We just have to add the base number (in decimal) we want and an `r` character to the beginning of the number (We use letters starting from A to represent digits greater than 9 while using a base greater then ten):

`8r312` (202 in decimal)

`16r312` (786 in decimal)

`16rABC` (2748 in decimal)

Scientific notation can be used on GNU Smalltalk, too. In scientific notation, we use an `e` character to represent *exponent of ten*. So the number we've written is multiplied by ten's exponent which is specified after the `e` character:

`1e2` (is equal to 100)

`0.3e5` (is equal to 30000)

`0.0015e4` (is equal to 15)

Fractions can also be represented on GNU Smalltalk by writing two integers around a slash character. Fractions are automatically simplified if they can be:

3/5

5/3

Now, we saw how to represent the numbers we will use, let's learn how to do math on those numbers. It is a good idea to start with basic mathematical operations.

+

We have a binary message named + for summation:

```
3 + 5
```

```
8
```

```
0.3 + 0.01
```

```
0.31
```

-

Hyphen is used for subtraction:

```
5 - 3
```

```
2
```

```
3 - 5
```

```
-2
```

*

`*` character (not `x` character) is used for multiplication:

```
2 * 3
```

```
6
```

```
0.1e-2 * 1e2
```

```
0.1
```

/

A slash character is used for division. GNU Smalltalk will give us a number in fraction or decimal representation whichever it decides is appropriate:

```
50/4
```

```
25/2
```

```
0.5/5
```

```
0.1
```

\\

Double backslash character is used for modulus operation:

```
5 \\ 3
```

```
2
```

```
3 \\ 3
```

```
0
```

between:and:

We can control whether a number is between other numbers or not by using `between:and:` selector. For example, to control if `3` is between `1` and `5`:

```
3 between: 1 and: 5
```

```
true
```

As you can see GST returned an object named `true`. This is an instance of `Boolean` class which is used for determining the accuracy of a logical statement. Then, how is logical inaccuracy represented? Let's look at the result of this code:

```
3 between: 4 and: 6
```

```
false
```

As you can see it is represented by an object named `false`. We will see how to use this `Boolean` objects more detailed on *Chapter 4, Controlling the Flow of Execution*.

abs

We use `abs` message to get the absolute value of a negative number:

```
-3 abs
```

```
3
```

degreesToRadians

We can convert a magnitude in degrees to a magnitude in radians by using `degreesToRadians` message. This will be useful for doing some trigonometric calculations:

```
180 degreesToRadians:
```

```
3.141592653589793
```

GST calculated pi number quite accurately and quite fast, didn't it?

cos

We can calculate cosine of an angle with `cos` message:

```
180 cos
```

```
-0.5984600690578581
```

Wait a minute, isn't the cosine of `180`, `-1`? Be careful of units. `cos` message wants to be sent to a magnitude of radians not degrees. So, you have to first convert it into radians with `degreesToRadians`. You will see how to use the result of `degreesToRadians` message to be sent `cos` message later.

negated

We use `negated` message to find out the negative of a number:

```
3 negated
```

```
-3
```

Watch out! If you send this message to an already-negative number then you will get the positive of that number:

```
-3 negated
```

```
3
```

raisedTo:

To find out the *n*th power of a number, we use `raisedTo:` message:

```
3 rasiedTo: 4
```

```
81
```

As you know, `0`th power of a number is always `1`:

```
3 raisedTo: 0
```

```
1
```

GST can also calculate negative powers of a number by returning nice fractional numbers:

```
3 raisedTo: -2
```

```
1/9
```

squared

We can find out the square of a number by using `squared` message instead of using `raisedTo: 2`:

```
3 squared
```

```
9
```

even

We can determine if a number is even or not by using the `even` message:

```
4 even
```

```
true
```

```
2222222221 even
```

```
false
```

odd

...And we can test if a number is odd or not by using the `odd` message:

```
3 odd
```

```
true
```

sign

`sign` message returns `1` if the number is positive and `-1` if the number is negative:

```
6 sign
```

```
1
```

```
-0.3
```

```
-1
```

Now, we will see some messages you can use with floating point numbers.

integerPart

integerPart message gives us the integer part of a floating point number:

```
0.7 integerPart
```

```
0.0
```

```
3.1 integerPart
```

```
3.0
```

truncated

To truncate a decimal number we use `truncated` message:

```
17.2 truncated
```

```
17
```

Note that this is not rounding:

```
17.6 truncated
```

```
17
```

To round decimal numbers we use...

rounded

`rounded` message:

```
17.6 rounded
```

```
18
```

Let us see some messages special to fractional numbers.

denominator

To get the denominator of a fractional number we use `denominator` message:

```
(3/4) denominator
```

```
3
```

Notice that we parenthesized the expression `3/4`. The reason is that if we didn't do that then the unary message `denominator` would be calculated before the expression `3/4` because, as we said before, unary messages are evaluated before binary messages.

numerator

We use numerator message to get the numerator of a fractional number:

```
(3/4) numerator
```

```
4
```

setNumerator:setDenominator:

To set the numerator and the denominator of a fraction, we use `setNumerator:setDenominator` message:

```
3/4 setNumerator: 5 setDenominator: 6
```

```
5/6
```

Note that, this time we didn't need to use parenthesis because this time we send a keyword message to the expression `3/4` and binary messages already have a precedence over keyword messages.

Character

Characters are the single symbols we use to represent data. For example; letters, special characters like %, and digits like 9 are all characters. They are individual objects in Smalltalk. Though, to tell Smalltalk environment that we want to treat a symbol as a character, we have to put a `$` (dollar sign) in front of it. For example:

```
$a

$%

$9
```

And we use the same methodology to represent dollar sign itself:

```
$$
```

We have a lot of methods to apply on characters, some of which are mentioned below.

asLowercase

If you have a letter to convert into lowercase then you can use `asLowercase` message:

```
$D asLowercase
```

```
$d
```

asUppercase

Uppercase counterpart of `asLowercase` is `asUppercase`.

```
$d asUppercase
```

```
$D
```

isAlphaNumeric

If you have a suspicion whether a symbol is alphanumeric, then you can use `isAlphaNumeric` message:

```
$% isAlphaNumeric
```

```
false
```

```
$3 isAlphaNumeric
```

```
true
```

You might need such methods when you expect some input from user, for example. Since you cannot know what a person might enter as data you should use this kind of methods to ensure that data entered is in a valid format.

isDigit

Maybe you just want to determine if a symbol is a digit:

```
$4 isDigit
```

```
true
```

```
$c isDigit
```

```
false
```

isLetter

Or you might be interested in whether the data is a letter or not:

```
$c isLetter
```

```
true
```

```
$4 isLetter
```

```
false
```

String

Strings are object compound of several character objects. We use strings to represent words, sentences, paragraphs etc. To tell Smalltalk environment that we are creating a string we put ' (single quote) character before and after a sequence of characters. For example:

```
'Canol'
```

```
'I want a cup of coffee'
```

are all string objects. We use two single quote character, consecutively, to represent a single quote character in a string:

```
'Eiffel Tower''s height varies as much as six inches, depending on
the temperature.'
```

Note that it is two single quote and not a double quote.

Since strings are used by programmers heavily, we have a lot of useful methods for them. Below are some messages I selected. Editor's Choice Awards go to:

includes:

If you wonder whether a string object includes a specific character object in it, you can use `includes:` message:

```
'Canol' includes: $n
```

```
true
```

Note that this message is case-sensitive:

```
'Canol' includes: $N
```

```
false
```

indexOf:

Sometimes, just knowing whether a string includes a character is not enough. You also want to know exactly where it is. You can use `indexOf:` message to learn it:

```
'Canol' indexOf: $n
```

```
3
```

If a character is not present in the string then this method returns `0`. And if character exists more than one time then it returns the index of the first appearance of that character in the string.

reverse

This is one of my favorite messages in Smalltalk. You can reverse a string via `reverse` message:

```
'.dlrow eht ni remmargorp tseb eht si lonaC' reverse
```

```
'Canol is the best programmer in the world.'
```

Isn't this fun?

countSubCollectionOccurrencesOf:

You can count how many occurrences of a certain string are there in another string via `countSubCollectionOccurrencesOf:` message:

```
'Thomas Edison, the inventor of the light bulb, was afraid of the
dark.' countSubCollectionOccurrencesOf: 'the'
```

```
3
```

,

If you want to concatenate two strings into one `String` object you can place a `,` (comma) character between them:

```
'Best', ' friends'
```

```
'Best friends'
```

Actually you can concatenate as many strings as you want via message chaining:

```
'Best', ' friends', ' should', ' never', ' be', ' separated.'
```

```
'Best friends should never be separated.'
```

asUppercase

You can convert a whole sentence into uppercase using `asUppercase` message:

```
'Hey I''m talking to you!' asUppercase
```

```
'HEY I''M TALKING TO YOU!'
```

And you can use `asLowercase` message to lowercase your sentence, though we won't give an example for that.

size

You can learn how many characters a string has by sending it the `size` message:

```
'There are 39 characters in this string.' size
```

```
39
```

Variables

> Variables in Smalltalk is a lot like variables we use in mathematics. You may keep that in mind while reading this topic to imagine this concept easier.

While writing a program, we will have a lot of objects. They will be created, manipulated and destroyed. But after creating an object, how can we refer to it at any place in our program? Suppose that we have a `Human` class (not a class provided by the standard library of GNU Smalltalk but an imaginary one) and we create new instances from it. Like humans in our world, they need a name so that we can communicate with them. Remember that we communicate with objects via sending messages to them and before message, we should reach that object to specify which object we want to be the receiver. This can be achieved with an expression which returns the object or referring the name of the object.

Suppose we have a `Human` instance named `Carl`. We can send him a message like:

```
Carl tellUsYourLastName
```

Here `Carl` refers to a `Human` instance.

Giving a name to an object is called *assignment* and we call this names *variable*s in programming jargon. This is because although a name refers to only one object, the object it refers to can be changed any time in the program. Before learning how to assign a name to an object, first discuss why would we want to do such a thing. Most probably, in an example like above, which we created a human, you wouldn't want to change the referred object. But let us take another example: suppose that we have a program which tracks the number of customers who visits a restaurant in a day. This program should increment the number by one every time a customer comes in. How can we achieve such a mission? Well, first we create a variable which refers to object `0`. Than every time a customer comes in; we call that variable, get the object it refers, sum it with `1` and assign it back to that variable. So, every time a customer comes in; the object, variable refers to, is changing.

Now it is time to learn the syntax for doing assignment. Before doing any assignments, we should create variables. The general structure for creating variables is:

```
| aVariable anotherVariable andAnotherVariable |
```

Here we created three variables named `aVariable`, `anotherVariable` and `andAnotherVariable`. We put the variable names we want to create between pipe characters. We separate variables with spaces and can create as many variable as we want. The general syntax for assignment is like this:

```
aVariableName := anObject
```

We use an operator called *assignment operator* which is composed of a colon and an equal sign without any whitespace in between. This operator assigns the object on its right to the variable on its left.

Every object created occupies space in computer memory and the task of variables is actually to hold the memory address of an object.

Getting User Input

In this section, we will introduce a code to obtain information from user via standard input device (most probably keyboard). If you ever used a console program before then you probably faced "Are you sure you want to do this? (y/n)" kind of questions which expect you to enter *y* or *n*. This is the technique used for getting that kind of user inputs from terminal.

```
"user_input.st"
"A program to demonstrate how to get input from user."

| userName |

Transcript show: 'What is your name? '.
userName := stdin nextLine.

Transcript show: ('Hello ', userName, '!'); cr.
```

```
What is your name? Canol
Hello Canol!
```

This program asks user for his/her name and displays a hello message including the name entered. The part which gets the user input is `userName := stdin nextLine`. `stdin nextLine` expects a keyboard input which is terminated by a newline character. The user, in this case, enters whatever he/she wants and presses <Enter> key to terminate the input process. `stdin nextLine` returns the input string, so we assigned it to the variable `userName`. Assignment operator has a lower precedence compared to sending any kind of message to an object, so first `stdin nextLine` is executed and then the assignment.

Common Classes and Their Usage: Part II

Now we will introduce some other useful classes you will probably use, frequently. We will also use the variable concept in our examples.

Array

We often need to group some kind of data together and do some tasks on them altogether. Usually, the data we grouped is connected in some way. Maybe a grocery list, the movies we like, the books we read etc. So we need an object that is capable of holding a collection of other objects. `Array`, `Set` and `Dictionary` classes are all capable of holding several objects together, in different ways.

The first one we will see is the `Array` class. This class holds a limited number of objects, in an order.

new:

You can create a new `Array` object by sending `new:` message to the `Array` class. You should supply a number to determine how many objects the array will be able to hold. The code below shows an example:

```
| anArray |


anArray := Array new: 10
```

```
(nil nil nil nil nil nil nil nil nil nil )
```

You can enter the lines above, respectively or write them into a file and then execute it via sending the script file to the GST interpreter. Please enter the lines as we write here because we will take advantage of the GNU Smalltalk memory so that we don't have to create a variable over and over.

Now, our `anArray` variable refers to an `Array` object which can hold ten other objects. If you entered above lines respectively, you should have got an output like above. It shows the current state of our `Array` object. At first, all of the objects of an `Array` object are initialized as `nil` object which is a special object to represent nothing.

at:

To get a single object at a certain position we can use `at:` message. Please keep entering the commands like we write here:

```
anArray at: 1
```

```
nil
```

We get the first object in our array which is, of course, the `nil` object. The positions of an array are called *index*es. In Smalltalk indexes begin with the number 1.

> For experienced programmers:
>
> In contrast to C based languages, in Smalltalk, indexes begin with 1, not with 0.

at:put:

To place an object to a certain position we use `at:put:` message:

```
anArray at: 1 put: 'Toothbrush'
```

```
'Toothbrush'
```

We can see what the objects in our array are by simply writing its name:

```
anArray
```

```
('Toothbrush' nil nil nil nil nil nil nil nil nil )
```

As you can see, now, the first object in our array is the `String` instance `'Toothbrush'`. Let's put a soap to our array:

```
anArray at: 2 put: 'Soap'
```

```
'Soap'
```

```
anArray
```

```
('Toothbrush' 'Soap' nil nil nil nil nil nil nil nil )
```

includes:

We can investigate if an array has a certain object in it by sending it the `includes:` message along with the object we are searching:

```
anArray includes: 'Soap'
```

```
true
```

```
anArray includes: 'Toothpaste'
```

```
false
```

reverse

Remember, we said that `Array` objects are ordered collections. There are ways to manipulate this order. One of them is to reverse it via `reverse` message:

```
anArray reverse
```

```
(nil nil nil nil nil nil nil nil 'Soap' 'Toothbrush' )
```

Set

`Set`s are a lot like `Array`s except that they do not hold their content in any order and they don't have any predefined limits like `Array`s. A `Set` is the GNU

Smalltalk counterpart of the sets in mathematics so it behaves much like the sets in mathematics.

new

We create a `Set` instance again with `new` message but without an argument this time:

```
| aSet |


aSet := Set new
```

```
Set ()
```

The output shows us that initially, our set does not contain any object in it.

add:

We can add an object into a `Set` by using `add:` message:

```
aSet add: 'Toothbrush'
```

```
'Toothbrush'
```

```
aSet
```

```
Set ('Toothbrush' )
```

Now, our set holds a `String` object which has the value of `'Toothbrush'`. Now let's add another `String` object:

```
aSet add: 'Soap'
```

```
'Soap'
```

```
aSet
```

```
Set ('Soap' 'Toothbrush' )
```

As you can see GST put it before the `'Toothbrush'` object. Actually, it didn't put it before, it just put it into the set, randomly. The place of `'Soap'` or `'Toothbrush'` object is unpredictable, in other words they don't have any certain placement inside the `Set`.

Question:

Can you tell why `Set` class does not have a `reverse` method?

Maybe one more example may help you understand it better:

```
aSet add: 'Toothpaste'
```

```
'Toothpaste'
```

```
aSet
```

```
Set ('Soap' 'Toothpaste' 'Toothbrush' )
```

Now, the last object we added went in between the other two old objects. But again, this is just a representation, actually they don't have any order.

There is an interesting result of not having an order inside `Set` object. When you try to add the same object twice into a `Set` object, you end up having it only once:

```
aSet add: 'Toothbrush'
```

```
'Toothbrush'
```

```
aSet
```

```
Set ('Soap' 'Toothpaste' 'Toothbrush' )
```

What does matter to the GST is if an object is in the `Set` or not. As you can see this is much like the sets in mathematics.

remove:

To remove an element from a `Set` you can use the `remove:` message:

```
aSet remove: 'Toothpaste'
```

```
'Toothpaste'
```

Note that `remove:` message didn't return the `Set` object, instead returned the removed object.

```
aSet
```

```
Set ('Soap' 'Toothbrush' )
```

As you see we didn't use an index to reach and remove the element, instead we used the element itself because elements inside a set does not have an index.

Note that removing an element from a set does not necessarily delete that object from the system. The object just does not belong to that set anymore. Let's demonstrate that:

```
| anElement |
anElement := 'Perfume'
```

```
'Perfume'
```

We created a separate variable pointing to a `String` object. Now, add it to our set:

```
aSet add: anElement
```

```
'Perfume'
```

```
aSet
```

```
Set ('Soap' 'Toothbrush' 'Perfume' )
```

Let's remove it:

```
aSet remove: anElement
```

```
'Perfume'
```

```
aSet
```

```
Set ('Soap' 'Toothbrush' )
```

Now, it is removed from the `Set`, let's control whether our variable is still alive:

```
anElement
```

```
'Perfume'
```

As you see, it is still alive and well. The `remove:` message we sent to the `Set` didn't affect our variable and the `String` object it points.

Dictionary

The last class we are going to show you is called `Dictionary`. `Dictionary`s, again, provide a way to group a bunch of objects together like `Array`s and `Set`s.

`Dictionary`s are like `Array`s in that they keep their data associated with an index key, but the difference is that the keys don't have to be numbers. You can define any object as a key for a data.

And `Dictionary`s are like `Set`s in that they don't keep their data in any order and they don't have a predefined limitation how much object you can keep in it.

new

We create a dictionary with `new` message just like `Set`s:

```
| aDictionary |
aDictionary := Dictionary new
```

```
Dictionary (
)
```

The output shows us that, initially, our `Dictionary` does not contain any object in it.

at:put:

We can add an object into a dictionary by using at:put: message:

```
aDictionary at: 'Canol' put: 'Gokel'
```

```
'Gokel'
```

```
aDictionary
```

```
Dictionary (
	'Canol'->'Gokel'
)
```

Our dictionary now holds a string named `Gokel` associated with another string named `Canol`. Now, let's add another `String` object:

```
aDictionary at: 'Paolo' put: 'Bonzini'
```

```
'Bonzini'
```

```
aDictionary
```

```
Dictionary (
	'Canol'->'Gokel'
	'Paolo'->'Bonzini'
)
```

keys

We can get all the keys inside a `Dictionary` by using `keys` message. GST will return a `Set` object which includes all the keys:

```
aDictionary keys
```

```
Set ('Canol' 'Paolo' )
```

removeKey:

To remove an element from a `Dictionary` you can use the `removeKey:` message:

```
aDictionary removeKey: 'Canol'
```

```
'Gokel'
```

```
aDictionary
```

```
Dictionary (
       'Paolo'->'Bonzini'
)
```

`removeKey:` message like `remove:` message from `Set` returned the removed object.

Review Questions

1. What is an object, a message and a class?
2. What is a variable? Why do we need them?
3. What are the types of messages? Can you classify two messages for each of the type, among the messages we saw on section, *Common Classes and Their Usage*?
4. What type of numbers can we deal with in GNU Smalltalk? Give some example numbers for each type.
5. Write a program which gets a number from user and displays its cube.
6. Write a program which gets two numbers from the user separated by a space and displays their arithmetic average.

   (Hint: You will get an input like `3 4` from the user which is a `String` holding two numbers. Use `tokenize: aString` message on this string to separate the numbers from each other. `tokenize:` message separates a string into pieces, which are called *token*s, wherever it finds its argument inside the receiver and returns an `Array` which consists of all the tokens.)
7. Write a program which holds definitions of some concepts we saw in *Chapter 1* in a `Dictionary` object. The program should ask the user to enter a word and then display the corresponding definition.
8. What is the difference between `Array`s and `Dictionary`s? Why would we use an `Array` when there is a more featured and charismatic class named `Dictionary`?

An intellectual is a man who says a simple thing in a difficult way; an artist is a man who says a difficult thing in a simple way.

Charles Bukowski

Chapter 4: Controlling the Flow of Execution

The programs we have seen so far executed the codes line by line in the order we write them. But real life isn't like this. We have to grow up and make some decisions. And because programs are to mimic real life problems, they, as well, involve decisions. Decisions make the execution of the code lines out of order. Some lines of code are executed according to a decision and some lines may never executed at all. For example, you should get your coat on if the weather is cold or shouldn't if it is hot.

There is more. Controlling the flow of execution of our programs does not just made of decisions. There are also repetitions, a thing that is not present at real life but a huge reason of why programming languages are present. For example, you may want to do something thousands of times. The first sample that comes to our minds is mathematical problems. Consider we want to find all the prime numbers up to 100 billion. Trying this on paper with a pencil is obviously not good for your health. Instead, we develop an algorithm to find whether a number is prime or not and then apply it on all the numbers from 1 to 100 billion. Thanks to the hardware capabilities today, we can find the solution of such problems in milliseconds and, importantly, without a mistake.

So we divide controlling into two main types: Selective and repetitive. *Selective controlling* selects a part of our code to be executed according to a condition (decision). And *repetitive controlling* will allow us to execute a part of the code over and over.

Before going on with how to do selective or repetitive controlling with GNU Smalltalk, we will see a concept named *block* which we will extensively use along with controlling messages.

Blocks

*Block*s are objects to bring together some expressions and hold them together for later use. They are nothing but a series of code written between square brackets. This is an example of a block:

```
['Hello World!' printNl.
(3 + 7) printNl]
```

The key things are to remember that blocks are also objects and when you create a block, the code in it won't be executed right away. You should send it a message named `value`, like this:

```
['Hello World!' printNl.
(3 + 7) printNl] value
```

```
'Hello World!'
10
10
```

Note that the object `10` is printed twice because the last expression is returned as the result of the block which is the object `10`. Again, this is only the case when you are entering the codes above interactively via terminal.

As we will see in control expressions, blocks are very useful object structures. But before going on with control expressions we should also mention about a feature of blocks, block arguments.

Sometimes blocks may need additional data to evaluate the codes in it. We can then create one or more *block argument*s to pass as many additional data as we want The general structure of a block with arguments is:

```
[:blockArgument1 :blockArgument2 | block-expression-1. block-
expression-2]
```

The block arguments are at the beginning of a block and every argument is declared with a preceding colon. Then the argument part of the block is separated from the main content via a pipe character.

We will mostly use this kind of blocks with special messages which know how to use them. But if you ever need to use such a block manually then you should send it a message whose selector includes as many `value:` as the argument number. For example, for a block with three arguments you should use the selector `value:value:value:`.

Let's finish this subject with an example program:

```
"blocks.st"
"A program which involves a block with an argument."


| greetings |


greetings := [:platesOfCornFlakes | 'I have eaten ',
platesOfCornFlakes printString, ' plates of corn flakes this
morning!'].


('Hello ma! ', (greetings value: 3)) printNl.
```

```
'Hello ma! I have eaten 3 plates of corn flakes this morning!'
```

Here we first created a variable named `greetings` to hold a block we will create. Then we created a block object with an argument and assigned it to the variable. Note that when we used the argument inside the expressions, we didn't write the preceding colon. It is only written in declaration time and then omitted. Finally, we used this block with an expression `greetings value: 3`. Don't forget to parenthesize the necessary groups for the sake of the precedence rules.

Now, we are ready to go on with controlling expressions.

Selective Controlling

ifTrue:

Our first selective control message is `ifTrue:`. Its general structure is like this:

```
anObject ifTrue: [block-expression-1. block-expression-2]
```

When it is sent to a `Boolean` object, it executes the expressions inside the code block if it is `true`, ignores if it is `false`.

Let's give an example:

```
| ourVariable |

ourVariable := true.

ourVariable ifTrue: [
      'Our variable is true.' printNl.
]
```

```
'Our variable is true.'
```

We first created a variable called `ourVariable` and then set it as a `true` object. Then sent it a `ifTrue:` message. Because `ourVariable` is set as `true`, it executed the code block.

Question:

Please try the same code with `ourVariable` is set as `false` object.

ifFalse:

`ifFalse:` message is the same with `ifTrue:` message except that its code block is executed if the object refers to a `false` object.

ifTrue:ifFalse:

There will be times that you want to do something if your object is true and want to do something else if it is false. This message gives you the opportunity to do that in an easy way. The general structure is like this:

```
anObject ifTrue: [block-expression-1. block-expression-2] ifFalse:
[block-expression-3. block-expression-4]
```

Of course nothing can stop you if you want to write it in a better-looking shape like:

```
an-object
      ifTrue: [
             the-code-block-to-execute
      ] ifFalse: [
             the-code-block-to-execute
      ]
```

We will finish the selective control messages by giving an example for `ifTrue:ifFalse:` message. Here it is:

```
| ourVariable |

ourVariable := false.

ourVariable ifTrue: [
      'Our variable is true.' printNl.
] ifFalse: [
      'Our variable is false.' printNl.
]
```

```
'Our variable is false.'
```

Repetitive Controlling

Time has come to learn one of the main reasons to use programming languages or machines. Because machines do not get tired or do not make mistakes, we can order them to do the same thing millions of times and they won't only fulfill, they will fulfill it also very fast, mostly in milliseconds. So today, we cannot think some of the areas without getting help of computers. For example, military hardware should be fast to be able to do thousands of complex calculations in very limited time, accurately. Or some science branches like astronomy require to make calculations which are almost impossible to do with paper and pencil.

Smalltalk is not an exceptional language for this kind of purposes. We will now see the most commonly used messages to create repetitive expressions.

whileTrue:

The general structure of this message is like this:

```
aBlock whileTrue: anotherBlock
```

This message is designed to be sent to block objects. When it is sent, the receiver block object is evaluated and if the evaluated value is to be a `true` object, then the block sent as the argument is executed. Up to this point, it is very much like an `ifTrue:` message except that it is sent to a block object, not directly to a `Boolean` object. But the main difference appears from this point on. If the receiver object evaluated to `true` and the argument block is executed, then the receiver block is controlled once more. If it still evaluates to `true`, then the argument block is executed once more. This cycle will be run until the receiver block evaluates to `false`.

While loops are called *indefinite loop*s because generally it is not known how many times the loop will be executed. If it is known, then you should consider checking your design again because most probably the upcoming loop techniques are better for your case.

Let's give an example to understand this message better:

```
"average.st"
"A program which evaluates the sum of the numbers entered to
demonstrate the whileTrue: message."

| sum enteredIntegers lastEnteredInteger |

sum := 0.
enteredIntegers := 0.

[ lastEnteredInteger ~= -1 ] whileTrue: [
      Transcript cr; show: 'Please enter a number. To exit the
program enter -1: '.
      lastEnteredInteger := stdin nextLine asInteger.

      sum := sum + lastEnteredInteger.
      enteredIntegers := enteredIntegers + 1.
      Transcript show: 'The average of the numbers entered so far
is: ', (sum / enteredIntegers) printString; cr.
]
```

```
Please enter a number. To exit the program enter -1: 3
The average of the numbers entered so far is: 3

Please enter a number. To exit the program enter -1: 4
The average of the numbers entered so far is: 7/2

Please enter a number. To exit the program enter -1: 3432
The average of the numbers entered so far is: 3439/3

Please enter a number. To exit the program enter -1: -1
The average of the numbers entered so far is: 1719/2
```

This example demands user to enter a number and then calculates and prints the average of them. The only way to stop program is to enter the number `-1`. We hold the entered number in variable called `lastEnteredInteger` and in every loop we control whether it is `-1` or not at the part:

```
[ lastEnteredInteger ~= -1 ] whileTrue: [
      ...
]
```

The operator `~=` checks if the object on its left is *not* equal to the object on its right.

While we progress with different inputs, GNU Smalltalk virtual machine gives us the average in a fractional form.

to:do:

General structure of this message is:

```
aNumber to: anotherNumber do: aBlock
```

This structure starts from number `aNumber` and evaluates the `aBlock` object until it reaches `anotherNumber`. It increases one at every loop. `aBlock` object also includes a block argument so that programmer can use the current index in expressions of `aBlock`.

We mentioned that `whileTrue:` message is usually used with indefinite cases. `to:do:` on the contrary, is used when it is known how many loops we needed. That's why it is sometimes called *definite loop*.

For experienced programmers:

Being a definite loop, you might guess that this message is used in Smalltalk instead of `for` loop of C based languages.

Now let's look at this structure via an example:

```
"5_lines.st"
"A program which prints 5 lines to demonstrate the usage of to:do:
message."

1 to: 5 do: [:x |
      Transcript show: 'This is the ', x printString, '. line.';
cr.
]
```

```
This is the 1. line.
This is the 2. line.
This is the 3. line.
This is the 4. line.
This is the 5. line.
```

This program prints five lines to the output each indicating their own index. Note that we used a block argument, `x`, for getting the index of each loop. If we didn't write an argument for block object then we would get an error.

to:by:do:

Sometimes we might not want to increase the index of `to:do:` message one by one. There is an alternative message called `to:by:do:` for this purpose. Its general structure is like this:

```
aNumber to: anotherNumber by: step do: aBlock
```

It allows us to reach the number `anotherNumber` in steps of the number defined by `step`. For example, write the same example above, this time by using this alternative message to increase the index two by two. Also count until ten, this time, so that the loop does not end very quickly:

```
"tobydo.st"
"A program to demonstrate the usage of to:by:do: message."

1 to: 10 by: 2 do: [:x |
      Transcript show: 'This is the ', x printString, '. line.';
cr.
]
```

```
This is the 1. line.
This is the 3. line.
This is the 5. line.
This is the 7. line.
This is the 9. line.
```

You can see the effect from the output. This message can also be used to make counting backwards, like:

```
"tobydo_backwards.st"
"A program to demonstrate the backward capability of to:by:do:
message."

5 to: 1 by: -1 do: [:x |
      Transcript show: 'Oh my god! I''m counting backwards! This is
the ', x printString, '. line!'; cr.
]
```

```
Oh my god! I'm counting backwards! This is the 5. line!
Oh my god! I'm counting backwards! This is the 4. line!
Oh my god! I'm counting backwards! This is the 3. line!
Oh my god! I'm counting backwards! This is the 2. line!
Oh my god! I'm counting backwards! This is the 1. line!
```

This is the end of this chapter. In the next chapter we will continue our journey of learning the base concepts of object oriented programming and also learn how to create our own classes.

Review Questions

1. What is the ultimate goal of the control messages? What kind of control messages are there? Give a few examples for each type.

2. What are blocks? Why do we need them?

3. What are definite and indefinite loops? Specify one message for each type of loops.

4. Write the program in Chapter 2, Review Questions, which was displaying a diamond, like this:

```
  /\
 /  \
/    \
\    /
```

```
  \  /
   \/
```

again, using the concepts we have seen in this chapter. Write the program in a way that we can specify how many slashes a side has, easily, so that we can form smaller or bigger diamonds.

5. Write a program which controls if a given number is prime or not and displays the result. The program should continue asking new numbers until number -1 is encountered.

6. This one comes from the inspiration of 99-bottles-of-beer.net web site. Write a program which outputs the lyrics of the 99 Bottles of Beer song which can be found at http://99-bottles-of-beer.net/lyrics.html

7. Write a program twice which displays numbers from 1 to 10 by using definite and indefinite loop techniques. Which version feels the right one for this task and why?

8. Write a program which always asks user to enter a new number and calculates the arithmetic average of numbers entered that far by using either definite or indefinite loop technique. If the user wants to finish entering new numbers, she should enter the string `finish`.

9. Now, can you write the one-line program mentioned in Review Question 4 in Chapter 2?

My brain: it's my second favorite organ.

Woody Allen

Chapter 5: Objects, Messages and Classes: Part II

Encapsulation

Ever wondered how the electronic circuits inside your cell phone works? Ever *needed* to know that? Most probably not. Because, we don't need to know all the manufacturing details about our cell phone to use it. Just reading the manual is enough to use all of its features. So, we cannot know all details, all scientific laws, all properties of an object we use and we don't need to. Object oriented programming introduces *encapsulation* concept to apply this real life fact in our applications.

Smalltalk does not allow you to directly modify internal state of an object. Objects can only be communicated by sending messages to them. So, Smalltalk encapsulates the unnecessary information.

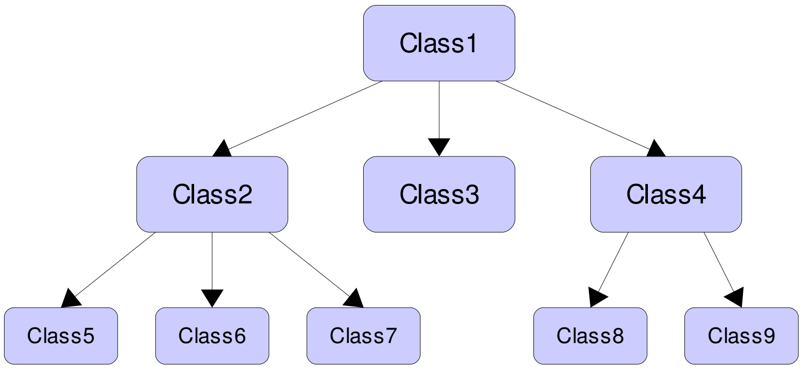
Beside encapsulation allows us to *think* in a real world manner, it also provides some kind of information hiding which helps our program to be solid and reusable. This means, if you hide an object's implementation from the programmer and provide him/her just a protocol to communicate with that object then the programmer won't be affected by implementation changes made in the future as long as the protocol is preserved.

For example, you created a `Calculator` class which is able to calculate the roots of a second degree equation. You just send the equation's information to it and it returns the roots... Maybe it uses an inefficient algorithm which causes it to calculate the roots in hours. But, well, it is better than nothing and you distributed the code to friends (we shall call them clients). Maybe, after distribution, you found a revolutionary approach which reduces the calculation time into seconds. The algorithm is changed but the protocol does not need to because the clients just need to provide the equality to the calculator and want it to solve the equation. How calculator solves the equation is not their business. So you can benefit the advantages of encapsulation and distribute your new code to clients. Clients just need to replace the `Calculator` class and does not need to change their own program.

Inheritance

Human beings like categorizing things. Because categorizing brings order and order brings easy-manipulation. Similar elements come together to form groups (or categories) and some special properties of them are the thing which distinguishes them from other group members. We may group some groups together, according to their similarities to form some other groups and also group them to form other groups. This process can go on until we reach one main group which is an ancestor of all others. This is actually a simulation of life itself.

The *elements* we mentioned above are *objects* and the *groups* are *classes*. What is the most general expression we use for elements? The answer is *object*. We can call everything as an object and then derive a special kind of object from it to which we give names like car, house, animal etc. But remember, we call objects which we use to create instances as *class*es. So, when we say car, house, animal we are actually talking about classes.



In this diagram each node represents a class. Some classes are derived from other classes. Branches are to represent them. For example; `Class2`, `Class3` and `Class4` are all derived from `Class1`. Also `Class5`, `Class6` and `Class7` are derived from `Class2`. And finally `Class8` and `Class9` are derived from `Class4`. We will make two definitions to represent this kind of relationships between classes.

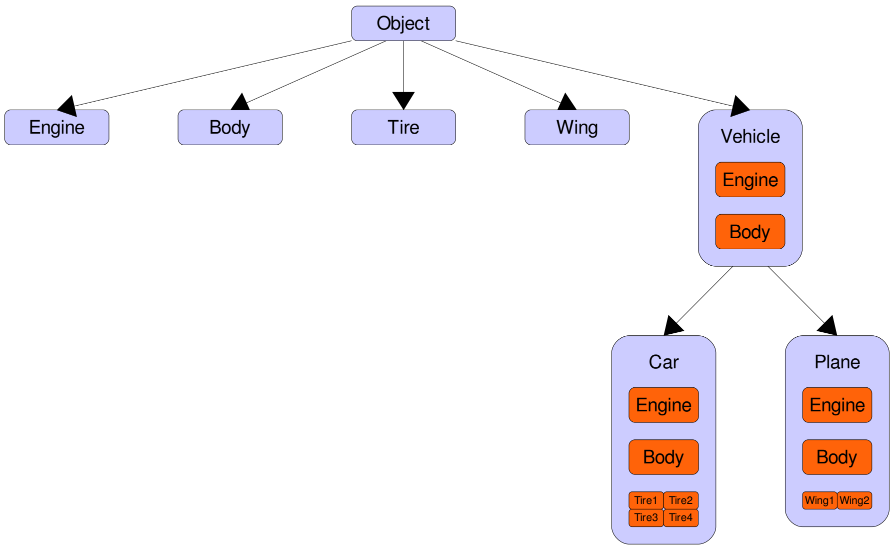
The classes derived from other classes are called *subclass*es of the class used for derivation. For example; `Class2`, `Class3` and `Class4` are all subclasses of `Class1`. Also `Class5`, `Class6` and `Class7` are all subclasses of `Class2`.

Every class is the *superclass* of its subclasses. For example; `Class1` is the superclass of `Class2`, `Class3` and `Class4`.

Derived classes (subclasses) *inherit* all the properties and behavior of their ancestors (superclasses).

Some classes in the above diagram have subclasses and every class has a superclass except Class1. Notice that classes may have more than one subclasses but they cannot have multiple superclasses.

An example figure is given below, which shows how to create a car and a plane in point of view of the object concept.



In this figure, each node represents a class. There are 8 classes, namely Object, Engine, Body, Wing, Tire, Vehicle, Car and Plane. Engine, Body, Wing and Vehicle are all subclasses of Object; Car and Plane are subclasses of Vehicle. A class may consist of instances of other classes. We see examples of that in the Vehicle, Car and Plane classes. Vehicle class has instances of Engine and Body classes. As Car and Plane classes are subclasses of Vehicle they inherited all the properties and behavior of their superclass, Vehicle. After inheriting the Engine and Body objects we added Car class four Tire objects and Plane class two Wing objects to distinguish them.

There are two definitions we should make in order not to confuse you about inheritance. There might be two kind of relations between objects. First, an object might have been derived from another object. This is called *is-a* relation. For example, Car is a Vehicle. Second, an object can consist of other objects. This is called *has-a* relation. For example, Vehicle has a Body.

The inheritance concept is related with is-a relation and not has-a relation.

Multiple Inheritance and Single Inheritance

Some programming languages allow programmers to create classes which inherits properties and behavior from more than one class. So, some classes have more than one superclass. This is called *multiple inheritance*. Other inheritance type, which allows only one superclass per class is called *single inheritance*.

The examples we have given so far are of single inheritance and the examples we will give from now on will also be of single inheritance because Smalltalk uses single inheritance to manage classes. It is still very arguable whether multiple inheritance is needed in object oriented design or only makes the inheritance more complex.

Polymorphism

In a world full of different kinds of objects, some objects may response the same message although they are different kind of objects, meaning they are instances of different classes. That's why some programming languages allow us to define methods with same selector in different classes. Or they allow a subclass to change a selector's behavior which it inherited from its superclass. This is called *polymorphism*.

Note:

The special case where a subclass changes the behavior of a method it inherited from its superclass is named as *overriding*.

We will now give a classical but very explanatory example for this concept. Think of a class called `Animal` and its two subclasses `Dog` and `Cat`. A method we would implement in `Animal` class would be `speak` because every animal communicates somehow with other livings by making some kind of noise (although dogs prefer to bite sometimes!). But when this message is sent, dogs would *bark* while cats would *miaow*. So they response to the same message but in a different way.

Creating Your Own Classes

GNU Smalltalk has hundreds of classes built-in but the universe has uncountable of them so GNU Smalltalk gives us the opportunity to create the classes we want for our needs. We will sometimes call the classes we made as *custom class*es.

First, we should talk about what classes are made of. Classes are the blue prints of objects so they definitely should describe all the things that objects have. For example, they have instance variables to define the properties of an object. They should have method definitions to make objects responsive to some certain messages. But they have more. There are *class variable*s. Class variables hold the information about the state of the class. There are also *class method*s which only classes can execute and not the instances of them.

The general syntax for creating classes is below:

```
SuperclassName subclass: SubclassName [
      | instanceVariable1 instanceVariable2 |

      classVariable1 := anObject.
      classVariable2 := anotherObject.

      <comment: 'Comment to describe our class'>

      SubclassName class >> aClassMethod: aParameter [
             "Comment to describe this class method"
             <category: 'Category of this class method'>

             | localVariable1 localVariable2 |

             ...

             ^objectToReturn
      ]

      anInstanceMethod [
             "Comment to describe this instance method"
             <category: 'Category of this instance method'>

             | localVariable1 localVariable2 |

             ...

             ^objectToReturn
      ]
]
```

Above pseudo-code shows how to create a class, add class and instance variables, add class and instance methods to it. We will explain it line by line but before that, we want to mention about some general concepts. Most of the things above are optional, for example you don't need to have class variables or methods. Or you don't need to write comments for anything or define the category of a method. But these are good software engineering habits and we recommend you to always write comments and define categories for methods.

Also another interesting thing in above code is the usage of white spaces. Some lines are *indent*ed by tabs or spaces, but why is that? It is just because for polishing the code for human eye, and there is no rule for how to use them. Mostly, we will indent a *body* of some new structure. For example, when we are writing the body of a class then we will indent it *one level* compared to the context. Interpretation of *one level* depends to the programmer. A programmer may choose as many spaces or tabs to insert as she wants for indenting a line but the important thing is to be consistent throughout the program so that it does not look messed up.

Other than the indentation, also most of the white spaces we used were programmer-dependent. You can choose not to place or even place more white spaces around anything. If it is a wrong usage (whenever it causes an ambiguity) then the interpreter will give you an error. For example, you cannot join the variable name and a keyword selector because interpreter cannot detect where the variable name ends and where the keyword selector begins. So you should put at least one space there.

Let's look at the code above line by line. The first thing we do is to define a class with the code piece below:

```
SuperclassName subclass: SubclassName [
        ...
]
```

The three dots in our code always mean that there are more expressions there but we cut it short to emphasize the really important part. So you should not put three dots in your code. With the part above, we told `SuperclassName` class to create a subclass named `SubclassName`. We did this to inherit the properties and behavior of the superclass so we don't have to create them again. After doing that, we opened a bracket to specify the class variables, instance variables, class methods and instance methods.

The part up to the opened bracket is called the *header* of the class. The part between the square brackets is called the *body* of the class.

The names we put between pipes are instance variables of our class:

```
        | instanceVariable1 instanceVariable2 |
```

Instance variable names are separated by white spaces. We have two instance variable here named `instanceVariable1` and `instanceVariable2`.

Then we define class variables by using assignment syntax:

```
        classVariable1 := anObject.
        classVariable2 := anotherObject.
```

`:=` is another message which makes the receiver an object referring to the argument object. So whenever we use `classVariable1` in our code from now on, the interpreter will understand that we are actually referring to the `anObject` object. We will use this assignment message also for changing the instance variables.

Mentioning about a variable for the first time is called *declaring a variable*. We should first declare a variable before being able to use it.

Next expression adds a comment to the class:

```
<comment: 'Comment to describe our class'>
```

They are only for documentation purposes and appear on IDEs we use for explaining what a class or method does so that we can get an idea of what it does without looking at its code which would took much longer and be harder. They do not affect the execution of the program, so, we can omit them.

And the time comes to defining the class and instance methods. We first created a class method:

```
SubclassName class >> aClassMethod: aParameter [
        "Comment to describe this class method"
        <category: 'Category-of-this-class-method'>

        | localVariable1 localVariable2 |

        ...

        ^objectToReturn
]
```

We again have a header part and a body part here. The header of the class definition is:

```
SubclassName class >> aClassMethod: aParameter
```

`SubclassName class >>` part of this header tells us that we are going to create a method for the class, not for the instances of the class. If we'd omitted this part then the method would be a part for the instances of this class. After this part we specify our method's message. The selector of this message is `aClassMethod:`. It takes a parameter named `aParameter`. *Parameter*s are the names given to the argument names when they are used in the header of methods, so these words are actually not that different from arguments and sometimes used in exchange. We may use the parameter names inside our method to refer to the argument passed to the method.

After specifying the selector we opened the bracket to define the method content which is called the body of the method:

```
        [
                "Comment to describe this class method"
                <category: 'Category of this class method'>

                | localVariable1 localVariable2 |


                ...


                ^objectToReturn
        ]
```

The content of a method can be made of a comment, a category, some local variables and the expressions to execute which are shown as three dots. After writing a comment with:

```
                "Comment to describe this class method"
```

we specify the category of this method with:

```
                <category: 'Category of this class method'>
```

These were optional things. So we don't have to write a comment or specify a category for our method.

The body of our method does the crucial thing. We first declare our local variables:

```
                | localVariable1 localVariable2 |
```

It is the same as declaring some instance variables. Local variables are local to the method and cannot be used out of the method body.

Then we return the object represented by `objectToReturn` variable back to the caller with a ^ character in front of it:

```
                ^objectToReturn
```

This is called a return expression and will be the famous evaluated value of our message. For example when we do `2 + 3`, `5` is returned via such an expression. Returning a value using a return expression is also optional because of two reasons. First, the task you are trying to achieve might not need to get any object back. Like, printing some characters to the standard output. Second, a method returns the last evaluated expression in the method no matter if we wrote a return expression explicitly or not. But it is usually a good programming habit to explicitly return values so that the returned value is more obvious to the reader of the source code.

The last thing to talk about is the instance method we created:

```
        anInstanceMethod [
                "Comment to describe this instance method"
                <category: 'Category of this instance method'>

                | localVariable1 localVariable2 |

                ...

                ^objectToReturn
        ]
```

We won't mention about the details of this code because it is almost identical with class method except that it does not have any class name in its header which makes it an instance method. So, it will be present in all of the instance objects we create from this class.

Creating Objects from Classes

After creating a class, you can create objects from it by using the pseudo-code below:

```
ourObjectName := SubclassName new
```

We sent the message `new` to create an object from `SubclassName`. But wait! We didn't defined a method called `new`! Then how is it possible to use a method called `new`? This is related with the concept of inheritance. Every class and hence object has some methods defined by default and `new` is one of them. It provides a way for creating objects from classes and you don't need to define it for yourself every time, thanks to inheritance.

`ourObjectName` is a reference name which points to the newly-created object. It is also called a variable. Now, whenever we write `ourObjectName` somewhere in our program, the interpreter will know which object we are talking about. For example, if you remember, we have defined a method named `anInstanceMethod` to use with our objects. We can send this message to our object by writing an expression like this:

```
ourObjectName anInstanceMethod
```

We first, wrote our variable's name and then write the message we want to sent to it. This expression will return (or evaluate to) `objectToReturn`.

We learned how to use instance methods but how about using class methods? This expression will give an error:

```
ourObjectName aClassMethod
```

Because `aClassMethod` is not for the instances, it is for the class itself. We can execute a method for class like this:

```
SubclassName aClassMethod
```

Or we can get some help -again- from the inherited method `class` for reaching the class of our object and send it the message like this:

```
ourObjectName class aClassMethod
```

Here we used chained message concept. Note that because the messages are evaluated from left to right our expression turned into

```
SubclassName aClassMethod
```

first.

Example

Now let's write a real example to illustrate all the concepts we learned so far. In this example we will create a class for animals. A class named `Animal` will hold information belonging to every animal while two special classes, `Dog` and `Cat` derived from it will hold information special to that kind of animal:

```
"animal.st"
"A program which creates some animal classes to illustrate the
object oriented concepts."

Object subclass: Animal [
      | name |

      animalNumber := 0.

      <comment: 'A class for defining animals.'>
```

```
        Animal class >> setAnimalNumber: number [
                "A class method to set the animal number."
                <category: 'accessing'>

                animalNumber := number.

                ^animalNumber
        ]

        Animal class >> getAnimalNumber [
                "A class method to get the animal number."
                <category: 'accessing'>

                ^animalNumber
        ]

        setName: newName [
                "An instance method to set the animal's name."
                <category: 'accessing'>

                name := newName.
        ]

        getName [
                "An instance method to get the animal's name."
                <category: 'accessing'>

                ^name
        ]
]

Animal subclass: Dog [
        <comment: 'A dog class.'>

        makeNoise [
                "An instance method to get the dog's noise."
                <category: 'accessing'>

                'Woof!' printNl.
        ]
]
```

```
Animal subclass: Cat [
      <comment: 'A cat class.'>

      makeNoise [
              "An instance method to get the cat's noise."
              <category: 'accessing'>

              'Miaow!' printNl.
      ]
]

dog := Dog new.
Animal setAnimalNumber: 1.
dog setName: 'Karabash'.
dog getName printNl.
dog makeNoise.

cat := Cat new.
Animal setAnimalNumber: 2.
cat setName: 'Minnosh'.
cat getName printNl.
cat makeNoise.
```

```
'Karabash'
'Woof!'
'Minnosh'
'Miaow!'
2
```

Now we will investigate this program line by line. We first create our class by inheriting an object named `Object` which is a special object: ancestor of all classes we will create. It comes with some useful methods predefined for us like `new`:

```
Object subclass: Animal [
      ...
]
```

Next thing we do is to declare an instance variable named `name` and a class variable named `animalNumber`. We will give a name to our animals and keep the number of the animals we created in the class variable:

```
        | name |

        animalNumber := 0.
```

For documentation purposes we write a class comment:

```
<comment: 'A class for defining animals.'>
```

Now, we start defining our methods which provides the behavior of our class. First, we create a class method to be able to *set* the animal number:

```
        Animal class >> setAnimalNumber: number [
                ...
        ]
```

Then we create a class method to *get* the animal number:

```
        Animal class >> getAnimalNumber [
                ...
        ]
```

This kind of set-get pairs appear frequently in programming and they are called *getter*s and *setter*s or *accessor*s.

After that we create our accessors for setting and getting the name of the animal:

```
        setName: newName [
                ...
        ]

        getName [
                ...
        ]
```

Now that we finished our first class, we will create two other classes which derives from it, called `Dog` and `Cat`:

```
Animal subclass: Dog [
        ...
]

Animal subclass: Cat [
        ...
]
```

Each class derives from Animal class so they already have `setName:` and getName methods. We don't need to (and should not) write them again. But we added method with selector makeNoise to each of them. We chose such a way because each of the classes make their own sound.

The rest of the code is for testing purposes. We create two objects and try out their methods by sending them appropriate messages:

```
dog := Dog new.
Animal setAnimalNumber: 1.
dog setName: 'Karabash'.
dog getName printNl.
dog makeNoise.

cat := Cat new.
Animal setAnimalNumber: 2.
cat setName: 'Minnosh'.
cat getName printNl.
cat makeNoise.
```

We first created a Dog object, then set the animal number to 1 and the name of the dog to 'Karabash'. Then we printed the name of our dog and made her bark.

Secondly, we created a Cat object and applied the same process we applied onto our dog. This time we chose the name 'Minnosh' and set the animal number to 2 because we now have two animals.

For experienced programmers:

Actually, the inheritance tree and other things like manually updating the animal number is of course a bad programming practice in real applications but we wanted to keep the sample simple.

The last statement merely prints the animal number which is kept inside our Animal class.

```
Animal getAnimalNumber printNl.
```

With this example we have seen, inheritance via the accessor methods. Accessor methods provided us communication with the inner status of the object which is composed of the class and instance variables. We've also seen polymorphism via the makeNoise method which behaves differently according to the class of the object we are sending the message to. And we've lastly seen the inheritance concept by observing the presence of the setName: and getName methods on every class which is derived from Animal class.

Extending and Modifying Classes

Suppose you are going to use a number's cube in your program, a lot. Wouldn't it be great to have a method in `Number` class named `cubed` so that we can send that message to get the 3rd power of the receiver? Programmers often need to do such modifications to a class and GNU Smalltalk provides a way to do it. The class might be a built-in one or a class written by the programmer.

So, you can add new methods to and modify existing methods in a class. General syntax to modify a class is:

```
className extend [
        | newInstanceVariable1 newInstanceVariable2 |

        methodSelector [
                  ...
        ]
]
```

Every instance variable you create will be added to the class.

If a method with the same selector has already been defined than the last one will *override* it and becomes the new implementation.

If a method with the same selector is not present, then it is added.

If you want to modify the class itself, then you should write `class` after the `className`, like this:

```
className class extend [
        | newClassVariable1 newClassVariable2 |

        methodSelector [
                  ...
        ]
]
```

Notice that this time we create class variables and class methods.

Now, let's create a `Human` class and then modify it in some ways. The core version of `Human` class will have some common properties and be able to answer to fundamental messages.

```
"human.st"
"The core version of the Human class."
```

```
Object subclass: Human [
      | name age |

      setName: aName [
             name := aName.
      ]

      getName [
             ^name
      ]

      setAge: anAge [
             age := anAge.
      ]

      getAge [
             ^age
      ]

      introduceYourself [
             Transcript show: 'Hello, my name is ', name, ' and
I''m ', age printString, ' years old.'; cr.
      ]

      > aHuman [
             ^age > aHuman getAge
      ]

      < aHuman [
             ^age < aHuman getAge
      ]

      = aHuman [
             ^age = aHuman getAge
      ]
]

| me myBrother |

me := Human new.
me setName: 'Canol Gökel'.
me setAge: 24.

myBrother := Human new.
myBrother setName: 'Gürol Gökel'.
myBrother setAge: 27.

me introduceYourself.
myBrother introduceYourself.

(me < myBrother) printNl.
```

```
Hello, my name is Canol Gökel and I'm 24 years old.
Hello, my name is Gürol Gökel and I'm 27 years old.
true
```

Our `Human` class is pretty simple. It has two instance variables named `name` and `age`. After defining the accessors (`setName:`, `getName` etc.) we defined a method named `introduceYourself`. When a Human object is sent this message, it introduces itself by telling some information about itself. We have also written methods to compare two `Human`'s age. If a comparison is wanted to be done among two `Human` objects, you can use >, < or = binary messages. Notice that we have used the instance variable age for the receiver and sent the `getAge` message to the argument object to reach the ages of them which are integer numbers. The comparison methods are already defined for `Integer` objects so we don't have to go deeper.

This implementation is pretty simple. Suppose that we needed to add a few more details to this class. We can modify the above code directly but there will be times that we don't want to modify the original code. For example, maybe more than one program is sharing the code and only one of them is requiring the extended version. Then keeping the original code would be a good idea. So, let's extend our `Human` class in our new test program. Copy and paste the code of older `Human` class to the indicated area:

```
"human_extended.st"
"A program which has an extended version of Human class."

... -> Put here the code of old Human class.

Human extend [
      | occupation experience |

      getOccupation [
              ^occupation
      ]

      setOccupation: anOccupation [
              occupation := anOccupation.
      ]

      getExperience [
              ^experience
      ]

      setExperience: anExperience [
              experience := anExperience.
      ]

      introduceYourself [
              Transcript show: 'Hello, my name is ', name, ' and 
I''m ', age printString, ' years old. I am ', occupation, '.'; cr.
      ]

      > aHuman [
              ^experience > aHuman getExperience
      ]

      < aHuman [
              ^experience < aHuman getExperience
      ]

      = aHuman [
              ^experience = aHuman getExperience
      ]
]

| me myFriend |

me := Human new.
me setName: 'Canol Gökel'.
me setAge: 24.
me setOccupation: 'an Engineer'.
me setExperience: 1.

myFriend := Human new.
myFriend setName: 'İsmail Arslan'.
myFriend setAge: 23.
myFriend setOccupation: 'an Engineer'.
myFriend setExperience: 3.

me introduceYourself.
myFriend introduceYourself.

 (me > myFriend) printNl.
```

```
Hello, my name is Canol Gökel and I'm 24 years old. I am an
Engineer.
Hello, my name is İsmail Arslan and I'm 23 years old. I am an
Engineer.
false
```

We have done a lot of things, let's look at them one by one. The first thing we did was adding two new instance variables named `occupation` and `experience` to keep the occupation of the human and its experience: on the job:

```
        | occupation experience |
```

Then we wrote the accessor methods for these new instance variables:

```
        getOccupation [
                ^occupation
        ]

        setOccupation: anOccupation [
                occupation := anOccupation.
        ]

        getExperience [
                ^experience
        ]

        setExperience: anExperience [
                experience := anExperience.
        ]
```

Because these accessors are not written before, they are *added* to the class' behavior. We then *rewrote* some methods:

```
        introduceYourself [
                Transcript show: 'Hello, my name is ', name, ' and
```

```
I''m ', age printString, ' years old. I am ', occupation, '.'; cr.
      ]

      > aHuman [
              ^experience > aHuman getExperience
      ]

      < aHuman [
              ^experience < aHuman getExperience
      ]

      = aHuman [
              ^experience = aHuman getExperience
      ]
```

They have overridden the old implementations. New introduction method includes the occupation of the `Human`. We also changed the comparison methods so that the comparison is being made according to the experiences of the `Human` objects, this time. This changes sound like they are made by an employment agency which uses `Human` class inside their applications.

self and super

It is time to learn two new keywords of GNU Smalltalk. The purposes of these two keywords we will see in a few seconds are the same: referring to the receiver object of the message while writing the class definition. The difference becomes apparent when we try to invoke a method on the referred object.

self

The first keyword to see is `self`. When we use this keyword in a class definition, it is treated as the receiver object. For example, we could define the basic `Human` class like this:

```
"human_self.st"
"The second version of the Human class written using self
keywords."

Object subclass: Human [
      | name age |

      setName: aName [
              name := aName.
      ]
```

```
	getName [
		^name
	]

	setAge: anAge [
		age := anAge.
	]

	getAge [
		^age
	]

	introduceYourself [
		Transcript show: 'Hello, my name is ', self getName, '
and I''m ', self getAge printString, ' years old.'; cr.
	]

	> aHuman [
		^self getAge > aHuman getAge
	]

	< aHuman [
		^self getAge < aHuman getAge
	]

	= aHuman [
		^self getAge = aHuman getAge
	]
]

| me myBrother |

me := Human new.
me setName: 'Canol Gökel'.
me setAge: 24.

myBrother := Human new.
myBrother setName: 'Gürol Gökel'.
myBrother setAge: 27.

me introduceYourself.
```

```
myBrother introduceYourself.


 (me < myBrother) printNl.
```

```
Hello, my name is Canol Gökel and I'm 24 years old. I am an
Engineer.
Hello, my name is İsmail Arslan and I'm 23 years old. I am an
Engineer.
false
```

The difference is that we used the `self` keyword to reach the attributes of our object instead of using them directly. For example, `self getName` means: "Send the message, `getName`, to the receiver object.".

The trick here is that the search for the method will begin directly at the class of the receiver object and continue to its ancestors until a definition is found. Remember this because it is the difference between `self` and `super` keywords.

We suggest you to use the accessors to handle the attributes like the above example, even if you are in the class definition and be able to reach the attributes, directly. This is the encapsulation property of object oriented design and a good programming practice because in real life you cannot know what is going to happen inside an object if you change something in it. For example, think that you have a `Molecule` object. When you add atoms in it, a lot of properties of the molecule might also change like the bond type, the geometrical shape etc. So you shouldn't just do `oxygen := oxygen + 1` you should use something like `addOxygenAtom`. which takes care of all the details that you cannot predict or remember.

Of course, at some point you have to use direct manipulation (like in setters or getters itself) but abstract as much as you can.

super

When writing custom classes and overriding a behavior we will sometimes need to refer to the ancestor's definition of the method inside our method. This might be the case when we just want to add some additional statements and not want to change the behavior completely. Let's see an example. In this example we will have a class for creating watches and we will have another class for water resistant watches derived from it:

```
"watches.st"
"A program which defines classes about watches to demonstrate the
usage of the super keyword."
```

```
Object subclass: Watch [
    | style chronometerCapability |

    <comment: 'A class for defining ordinary watches.'>

    setStyle: theStyle [
        "A method to set the style of the watch."

        style := theStyle.
    ]

    getStyle [
        "A method to get the style of the watch."

        ^style
    ]

    setChronometerCapability: theChronometerCapability [
        "A method to specify if the watch has chronometer
capabilities."

        chronometerCapability := theChronometerCapability.
    ]

    getChronometerCapability [
        "A method to determine if the watch has chronometer
capabilities."

        ^chronometerCapability.
    ]

    listYourFeatures [
        "A method to print out the features of the watch."

        Transcript show: 'Style: ', self getStyle; cr.
        Transcript show: 'Chronometer capabilities: ', self
getChronometerCapability printString; cr.
    ]
]

Watch subclass: WaterResistantWatch [
    | resistanceDepth |
    <comment: 'A class for defining water resistant watches.'>

    setResistanceDepth: aDepth [
        "A method to set the resistance depth of the watch."

        resistanceDepth := aDepth.
    ]

    getResistanceDepth [
        "A method to get the resistance depth of the watch."

        ^resistanceDepth.
    ]

    listYourFeatures [
        "A method to print out the features of the water resistant
watch."

        super listYourFeatures.

        Transcript show: 'Resistance depth: ', self
getResistanceDepth printString; cr.
    ]
]

| watch1 watch2 |

watch1 := Watch new.
watch1 setStyle: 'Analog'.
watch1 setChronometerCapability: true.

watch2 := WaterResistantWatch new.
watch2 setStyle: 'Digital'.
watch2 setChronometerCapability: false.
watch2 setResistanceDepth: 30.

Transcript show: 'The features of the 1st watch:'; cr.
Transcript show: '------------------------------'; cr.
watch1 listYourFeatures.

Transcript cr.
Transcript show: 'The features of the 2nd watch:'; cr.
Transcript show: '------------------------------'; cr.
watch2 listYourFeatures.
```

```
The features of the 1st watch:
------------------------------
Style: Analog
Chronometer capabilities: true

The features of the 2nd watch:
------------------------------
Style: Digital
Chronometer capabilities: false
Resistance depth: 30
```

In this program we started by defining a class named `Watch` to represent ordinary watches. It has some properties like `style` which determines whether the watch is analog or digital and `chronometerCapability` which determines whether the watch has chronometer capabilities. After defining the accessor methods for these properties we created another method named `listYourFeatures` to print out the features of the watch to standard output. So far, we didn't see anything unusual.

After defining the `Watch` class we derived a new class named `WaterResistantWatch` to represent, well, water resistant watches. We added a new property named `resistanceDepth` to determine the depth the watch can stand without leaking the water inside. Now, the tricky part comes: we overrode the `listYourFeatures` method of its superclass and put the line below to the beginning of the method:

```
        super listYourFeatures.
```

This sends the `listYourFeatures` message to itself but the difference of the keyword `super` from the keyword `self` is that it will start looking up the definition of the method starting from the ancestor of the class, which is the class `Watch` in this case. So the `listYourFeatures` method of `Watch` is invoked instead of the `listYourFeatures` method of `WaterResistantWatch` class.

Why do we do this? Because `Watch` class already has some of the properties `WaterResistantWatch` has and it already has the capability of listing those features. We can either copy and paste the same code into the new `listYourFeatures` method or we can call the method defined inside the superclass. The latter way is preferred by programmers because it reduces the repetition of the code which provides less work when refactoring your application.

For example, our code won't be affected whenever a new property is introduced to the `Watch` class because the changes are automatically reflected to the output by using the `super` keyword, otherwise we would have to copy the changed code into our `WaterResistantWatch` every time.

It would be erroneous if we used the `self` keyword instead, because it would call itself recursively and this would cause an infinite cycle. Also it wouldn't give us the effect we wanted because the code we need is written in superclass' definition not in the class we are doing implementation right now.

In the rest of the program we define two watches which are of `Watch` and `WaterResistantWatch` class and list their features after setting their properties. As we expected, the second watch displayed the properties it inherited from its superclass, as well as its special property `resistanceDepth`.

Review Questions

1. Explain polymorphism, inheritance, and encapsulation in a few sentences.
2. Now, think of symbol > as *better than* and not *greater than*. Create a `Man` class which inherits from the `Human` class we created before with additional instance variables `money` and `handsomeness`. You know for us, men, appearance is not important, the important thing is the beauty inside. So create the `Woman` class with additional instance variables `honesty` and `generosity`. All this instance variables will get an integer point over ten.

   Implement the > method in `Man` class such that it sums the points of `money` and `handsomeness` and returns `true` if the receiver's point is higher. Add an exception to this rule, which is, if the name of one of the `Man` objects is "Canol Gökel" then it is always the better `Man`.

   Implement the > method in `Women` class in a similar manner. Sum the points of `honesty` and `generosity`, than return a `true` or `false` object, accordingly. There is no need for an exception this time, after all, all the womens are the same...
3. Extend the `Number` class so that it can respond the `cubed` message by returning the cube of the receiver.
4. Explain the purpose of `self` and `super` keywords and their difference from each other.

One of my most productive days was throwing away 1000 lines of code.

Kenneth Thompson

Chapter 6: What to Do Next

How Can I Improve Myself?

Although Smalltalk is a small language syntactically, the things that can be done using it is almost unlimited and hundreds of the built-in classes and their methods are there to help you. We've covered almost all of the grammatical part of GNU Smalltalk and some of the most commonly used built-in classes of it. There are still many classes waiting for you to explore and the most effective way to do it is doing practice. In programming world practicing means simply writing programs. Maybe you already have a project on your mind but if you don't, then there are plenty of interesting problems that you can try to solve using GNU Smalltalk. Some web sites are given at section, Further Reading, later in this chapter.

Another important thing for improving yourself is always being in contact with other Smalltalkers so that you can be informed about latest news on Smalltalk world and also don't feel alone. Again some web site, mailing list and blog addresses are given at the end of this chapter.

These were generally advises to improve your practical side because you have been already learning the theoretical things of Smalltalk by reading these book. At some point you may get a feeling that you should also improve your theoretical knowledge, after all, this book is a work in progress and although it covers all the essentials needed to do practical programming, it lacks some general advanced programming concepts and some advanced subjects special to the Smalltalk language. Also looking at this beautiful language from different point of views would never be harmful. That's why we also gave some reading material at the next section.

Further Reading

Smalltalk-80: The Language and its Implementation by Adele Goldberg and David Robson

This is the official book which describes the Smalltalk-80 programming language and also mentions about how to implement it. This book is also known as the *Blue Book*. You can get a lot of details about Smalltalk programming language by reading this book.

GNU Smalltalk Documentation - http://smalltalk.gnu.org/documentation

This is where you can find further information about GNU Smalltalk. There is a user manual as well as library references.

Useful Sites

GNU Smalltalk - http://smalltalk.gnu.org

The official GNU Smalltalk site. You can get the latest version of GNU Smalltalk, news and documentation here. Also there is a blog section where developers write entries as well as other members of the site.

Smalltalk.org - http://www.smalltalk.org

A huge source for every kind of information about Smalltalk.

Stephane Ducasse :: Free Online Books - http://stephane.ducasse.free.fr/FreeBooks.html

A site which gives a lot of links to freely available books related with Smalltalk language.

Planet Smalltalk - http://planet.smalltalk.org

A blog planet where Smalltalk related blog entries from all over the internet is published. One of the best ways to keep you informed about Smalltalk world. You can also have your blog posts published on this site by posting to GNU Smalltalk blogs!

Project Euler - http://projecteuler.net

This is a site where there are several hundred programming problems you can find and people try to get a higher rank by solving this problems with any language they want. The problems need a little bit mathematics knowledge. The site is well organized and solving a problem is a lot of fun. You can improve your GNU Smalltalk knowledge and algorithm ability significantly by solving a few of these problems. Also you can contribute to improve the rank of the Smalltalk language!

Smalltalk Jobs - http://smalltalkjobs.dabbledb.com

If you are searching for jobs related with Smalltalk language then you can look at this page to get one.

Mailing Lists

Mailing lists are a great way to reach as many people as possible regarding to your problems/suggestions/thoughts. Just send an email to the mailing list and everybody who subscribed to the list will get your message and be able to reply.

You can go http://smalltalk.gnu.org/community/ml to get more information about how to subscribe to the mailing list or to search previously posted messages.

IRC

The fastest way to get help from GNU Smalltalk community is to ask your question at IRC (Internet Relay Chat). The address for GST in IRC world is #gnu-smalltalk channel at irc.freenode.net. You can use an IRC client like XChat on Linux or mIRC on Windows to connect to this channel.

Wait a minute...

Pope Alexander VI (Last words of him)

Appendix A: Installing Programming Environment

At the time of writing, GNU Smalltalk was distributed, mostly, via its source code. That means you download the source code of GNU Smalltalk, compile it and use it. If you are lucky, then you might have a binary distribution for your operating system. We will mention about all this possibilities in this chapter.

Installing GNU Smalltalk on Linux Platforms

There two main ways to install GNU Smalltalk into your Linux system. One of them is using the package manager which comes with your Linux distribution and the other one is to compile GNU Smalltalk from its source code.

Installing via a Package Manager

There are several package managers but to name a few of famous ones: it is named YUM in Fedora, Synaptic Package Manager in Ubuntu and PiSi on Pardus. So, simply search for Smalltalk with your package manager and install the GNU Smalltalk package with highest version number. A version number 3.0 or above would be fine but if it is as old as 2.x then I suggest installing from source code like explained below.

Installing via Compiling the Source Code

Package managers does not always have an up-to-date GNU Smalltalk package. Also it might be the case that your distribution does not come with a package manager at all. Then you don't have any choice but to compile GNU Smalltalk by yourself. Although it is not as easy as the first case, it is not that hard either. If you follow the steps we will give below, your GNU Smalltalk environment will be ready to use in a few minutes.

GNU Smalltalk is implemented in C and GNU Smalltalk programming languages. So you should have a C compilation environment installed first. There is a strong probability that you have them already installed but if it is not the case then you may get your system administrator to install them for you or get help from web page of your Linux distribution to learn how to get them installed. GNU Smalltalk uses some GNU tools to provide an easier compilation experience.

Here are the 10 golden steps for being successful in your life:

1. GNU Smalltalk has some extra packages which extends the capabilities of it. One of them is named *Blox* which provides us a way to create

graphical user interfaces with GNU Smalltalk. Also there is a very useful feature called *Class Browser* for GNU Smalltalk which requires Blox. But Blox package is dependent to two other software called Tcl and Tk. So we will first download the source codes of Tcl/Tk and compile them.

Enter the site below:

http://www.tcl.tk/software/tcltk/download.html

and download one of the Tcl and one of the Tk source code packages with tar.gz extension. In my case, I downloaded tcl8.4.19-src.tar.gz and tk8.4.19-src.tar.gz.

2. Extract the source code packages into a convenient folder.
3. Open your terminal and go to the *unix* folder under the Tcl source code folder where you extracted the Tcl source code into. In my case:

```
cd /home/canol/Desktop/tcl8.4.19/unix
```

4. Enter the following commands, respectively. This will compile the source codes of Tcl and make them ready to be used by GNU Smalltalk (This steps may take a few minutes):

```
./configure
make
make install
```

If you get an error after doing `make install` like "Permission Denied" or similar, this means you have to log in as super user before applying that command.

1. Open your terminal and go to the *unix* folder under the Tk source code folder where you extracted the Tk source code into. In my case:

```
cd /home/canol/Desktop/tk8.4.19/unix
```

2. Enter the following commands, respectively. This will compile the source codes of Tk and make them ready to be used by GNU Smalltalk:

```
./configure
make
make install
```

3. Now we are ready to compile GNU Smalltalk itself. Enter the site below:

ftp://ftp.gnu.org/gnu/smalltalk

and download the latest source code package. In my case, I downloaded smalltalk-3.1.tar.gz

4. Extract the source code package into a convenient folder.
5. Go to the GNU Smalltalk source code folder where you extracted the GNU Smalltalk source code into. In my case:

```
cd /home/canol/Desktop/smalltalk-3.1
```

6. Enter the following commands, respectively. This will compile the source codes of GNU Smalltalk:

```
./configure
make
make install
```

If you didn't get an error while doing these ten steps above, then you are now ready to do some coding.

Installing GNU Smalltalk on Windows Platforms

At the time of writing, there is unfortunately no Windows installer so to install GNU Smalltalk interpreter to your computer you should compile it from the source. GNU Smalltalk can be successfully compiled via MinGW system which you can find from:

http://www.mingw.org

But you will also need to download the dependencies of the extra packages and compile them before GNU Smalltalk. You can get a little bit more information about compiling GNU Smalltalk from "Installing GNU Smalltalk on Linux Platform" section of this chapter as compiling process in MinGW is alike compiling process on Linux.

However a process is going on and there is a possibility that a Windows installer will be published soon. You can get the installer from GNU Smalltalk official site:

http://smalltalk.gnu.org/download

as soon as it is released. So keep an eye on the official web site and/or send mails to the mailing list if there is a progress on the Windows installer.

Appendix B: ASCII Table

Number (Decimal / Binary)	Character	Number (Decimal / Binary)	Character	Number (Decimal / Binary)	Character	Number (Decimal / Binary)	Character
0 / 000 0000	NUL (Null Character)	32 / 010 0000		64 / 100 0000	@	96 / 110 0000	`
1 / 000 0001	SOH (Start of Header)	33 / 010 0001	!	65 / 100 0001	A	97 / 110 0001	a
2 / 000 0010	STX (Start of Text)	34 / 010 0010	"	66 / 100 0010	B	98 / 110 0010	b
3 / 000 0011	ETX (End of Text)	35 / 010 0011	#	67 / 100 0011	C	99 / 110 0011	c
4 / 000 0100	EOT (End of Transmission)	36 / 010 0100	$	68 / 100 0100	D	100 / 110 0100	d
5 / 000 0101	ENQ (Enquiry)	37 / 010 0101	%	69 / 100 0101	E	101 / 110 0101	e
6 / 000 0110	ACK (Acknowledgment)	38 / 010 0110	&	70 / 100 0110	F	102 / 110 0110	f
7 / 000 0111	BEL (Bell)	39 / 010 0111	'	71 / 100 0111	G	103 / 110 0111	g
8 / 000 1000	BS (Backspace)	40 / 010 1000	(	72 / 100 1000	H	104 / 110 1000	h
9 / 000 1001	HT (Horizontal Tab)	41 / 010 1001	)	73 / 100 1001	I	105 / 110 1001	i
10 / 000 1010	LF (Line Feed)	42 / 010 1010	*	74 / 100 1010	J	106 / 110 1010	j
11 / 000 1011	VT (Vertical Tab)	43 / 010 1011	+	75 / 100 1011	K	107 / 110 1011	k
12 / 000 1100	FF (Form Feed)	44 / 010 1100	,	76 / 100 1100	L	108 / 110 1100	l
13 / 000 1101	CR (Carriage Return)	45 / 010 1101	-	77 / 100 1101	M	109 / 110 1101	m
14 / 000 1110	SO (Shift Out)	46 / 010 1110	.	78 / 100 1110	N	110 / 110 1110	n
15 / 000 1111	SI (Shift In)	47 / 010 1111	/	79 / 100 1111	O	111 / 110 1111	o
16 / 001 0000	DLE (Data Link Escape)	48 / 011 0000	0	80 / 101 0000	P	112 / 111 0000	p
17 / 001 0001	DC1 (Device Control 1)	49 / 011 0001	1	81 / 101 0001	Q	113 / 111 0001	q
18 / 001 0010	DC2 (Device Control 2)	50 / 011 0010	2	82 / 101 0010	R	114 / 111 0010	r
19 / 001 0011	DC3 (Device Control 3)	51 / 011 0011	3	83 / 101 0011	S	115 / 111 0011	s
20 / 001 0100	DC4 (Device Control 4)	52 / 011 0100	4	84 / 101 0100	T	116 / 111 0100	t
21 / 001 0101	NAK (Negative Acknowledgment)	53 / 011 0101	5	85 / 101 0101	U	117 / 111 0101	u
22 / 001 0110	SYN (Synchronous Idle)	54 / 011 0110	6	86 / 101 0110	V	118 / 111 0110	v
23 / 001 0111	ETB (End of Transmission Block)	55 / 011 0111	7	87 / 101 0111	W	119 / 111 0111	w
24 / 001 1000	CAN (Cancel)	56 / 011 1000	8	88 / 101 1000	X	120 / 111 1000	x
25 / 001 1001	EM (End of Medium)	57 / 011 1001	9	89 / 101 1001	Y	121 / 111 1001	y
26 / 001 1010	SUB (Substitute)	58 / 011 1010	:	90 / 101 1010	Z	122 / 111 1010	z
27 / 001 1011	ESC (Escape)	59 / 011 1011	;	91 / 101 1011	[	123 / 111 1011	{
28 / 001 1100	FS (File Separator)	60 / 011 1100	<	92 / 101 1100	\	124 / 111 1100	|
29 / 001 1101	GS (Group Separator)	61 / 011 1101	=	93 / 101 1101	]	125 / 111 1101	}
30 / 001 1110	RS (Record Separator)	62 / 011 1110	>	94 / 101 1110	^	126 / 111 1110	~
31 / 001 1111	US (Unit Separator)	63 / 011 1111	?	95 / 101 1111	_	127 / 111 1111	DEL (Delete)

Appendix C: Answers of Review Questions

Chapter 1

1. Because of the hardware design, computers can only understand two states. We represent this two states with digits 0 and 1. Combinations of zeros and ones generate some commands that computer hardware understands. We call these command groups which provide a communication between us and computer hardware as programming language.

   We need them to communicate with computer hardware, because computer hardware can understand only two states and the programming language we use is eventually translated into the sum of combinations of this two states which is called machine language.

2. Programs written in a compiled language should be translated into machine language by other programs called compiler before the execution by the user.

   Programs written in an interpreted language is converted to machine code during the execution time by other programs called interpreter. They do not need a compilation process before they can be executed by the user but they are a little bit slower than compiled machines, though this is mostly not a problem anymore for modern computers and for not-so-performance sensitive projects.

   Although Smalltalk is designed as a language which uses a virtual machine to work, GNU Smalltalk acts more like an interpreted language.

3. A programming paradigm is how a programming language looks at the problems to be solved.

4. Yes. Such programming languages are called multi-paradigm programming languages.

   No, GNU Smalltalk allows programmers to use only object-oriented programming paradigm.

5. **From decimal to binary:**

$$543/2=271 \quad (remainder\,\mathbf{1})$$

$$271/2=135 \quad (remainder\,\mathbf{1})$$

$$135/2=67 \quad (remainder\,\mathbf{1})$$

$$67/2=33 \quad (remainder\,\mathbf{1})$$

$$33/2=16 \quad (remainder\,\mathbf{1})$$

$$16/2=8 \quad (remainder\ \mathbf{0})$$

$$8/2=4 \quad (remainder\ \mathbf{0})$$

$$4/2=2 \quad (remainder\ \mathbf{0})$$

$$2/2=\mathbf{1} \quad (remainder\ \mathbf{0})$$

So the answer is: **1000011111**

From decimal to octal:

We can use the same method to convert it from base-10 to base-8 but we learned an easier way, so let's use it:

$$(543)_{10}=\underbrace{001}_{1}\,\underbrace{000}_{0}\,\underbrace{011}_{3}\,\underbrace{111}_{7}=(1037)_8$$

From decimal to hexadecimal:

$$(543)_{10}=\underbrace{0010}_{2}\,\underbrace{0001}_{1}\,\underbrace{1111}_{F}=(21F)_{16}$$

6. **From binary to decimal:**

$$(10110100)_2=0\cdot 2^0+0\cdot 2^1+1\cdot 2^2+0\cdot 2^3+1\cdot 2^4+1\cdot 2^5+0\cdot 2^6+1\cdot 2^7$$
$$(10110100)_2=0+0+4+0+16+32+0+128=(180)_{10}$$

From binary to octal:

$$(010110100)_2=\underbrace{010}_{4}\,\underbrace{110}_{6}\,\underbrace{100}_{4}=(464)_8$$

From binary to hexadecimal:

$$(10110100)_2=\underbrace{1011}_{B}\,\underbrace{0100}_{4}=(B4)_{16}$$

7. **From hexadecimal to binary:**

$$(A93F)_{16}=\underbrace{1010}_{A}\,\underbrace{1001}_{9}\,\underbrace{0011}_{3}\,\underbrace{1111}_{F}=(1010100100111111)_2$$

From hexadecimal to octal:

We can look at place value of each digit, treat them as octal numbers and sum them up again in octal rules but this is hard for humans because we used to and tend to treat numbers as they are decimal so converting a hexadecimal number to binary and converting it to octal is usually simpler:

$$(A93F)_{16}=\underbrace{001}_{1}\,\underbrace{010}_{2}\,\underbrace{100}_{4}\,\underbrace{100}_{4}\,\underbrace{111}_{7}\,\underbrace{111}_{7}=(124477)_8$$

From hexadecimal to decimal:

Now, we can use the method we mentioned above:

$$(A93F)_{16} = 15 \cdot 16^0 + 3 \cdot 16^1 + 9 \cdot 16^2 + 10 \cdot 16^3$$
$$(A93F)_{16} = 15 + 48 + 2304 + 40960 = (43327)_{10}$$

8. Binary files are composed of zeros or ones placed according to a format specification. They can specify any kind of information from a document to image or video. Text files are however specially designed binary files to contain only character data using a character set and encoding.

   .zip and .avi files may be given as examples of other binary files.

   You can try to open a file with a text editor and if it contains only meaningful characters and not funny symbols all around, it is most probably a text file. Also some text editors might detect that a file is not written in a recognized text encoding and warn the user about it.

9. Text editors are designed only to work with text files while word processors have most probably have their own binary file format to save documents. Like .doc for Microsoft Word and .odt for OpenOffice.org Writer. Word processors have a lot of fancy formatting tools to write a text in bold, italic, colored etc. which we don't use when writing text files.

   Most word processors have options to work with text files but they have so much unnecessary features we won't use that using them is like using a limousine to harvest.

Chapter 2

1.

```
"answer_2_1.st"

Transcript show: '  /\  '; cr.
Transcript show: ' /  \ '; cr.
Transcript show: '/    \'; cr.
Transcript show: '\    /'; cr.
Transcript show: ' \  / '; cr.
Transcript show: '  \/  '; cr.
```

```
  /\
 /  \
/    \
\    /
 \  /
```

```
  \/
```

2. When we have a small task to complete or when we are experimenting things, using terminal to enter the program instructions is very practical. But if we have to write a big program, saving it to hard disk by writing its source code into a text file will make it easy for us to work on it later on. Also saving the source code provides us the opportunity to use that program again whenever we want.

3. Comments provide the reader of the source code additional information so that he/she understands the code faster. The reader might be another programmer or the programmer who wrote that program himself. Even we might not remember what we intended to do with a specific part of the source code when reading it a few weeks later. Also in GNU Smalltalk they are used for documentation purposes which we will see later.

   In GNU Smalltalk whatever we write between double quotes are treated as comments by interpreter, except double quotes itself. If we want to write double quotes in a comment, we should write two double quotes, consecutively or we might use single quotes if appropriate for reader.

4. We can do that with our current knowledge by writing every single number manually like:

```
1 printNl.
2 printNl.
3 printNl.
.
.
.
998 printNl.
999 printNl
1000 printNl.
```

   But of course, this is not practical and nobody can force us to do that. Later on this book, we will learn about *loops* and see how we can achieve this task easily in just one line of code.

   If we imagine about it, we will probably come to a point that: "We should be able to define a *range* and make the `printNl` command act on that range.", which is the exact solution we will see.

5. Everything the computer generously give us is an output. It can be an image, a sound, a movement etc. So, actually, beside our monitor image which is in the

form of a terminal text in this case, our speakers or a printer may also be classified as output devices.

Also, keyboard is not the only input device. Every device which give us the opportunity to send information to the computer is an input device. For example, a mouse, a scanner, a web cam, a touch screen may all be specified as input devices.

Programming languages usually give us to change standard input and output devices so that the computer and the user interacts with each other using different ways and GNU Smalltalk is not an exception.

Chapter 3

1. An object is a unit which represents a *thing* in the program. It has some definitions how to respond to certain messages it receives from outside (usually from us or from other objects) which are called methods. We communicate with objects via messages we send to invoke the methods inside them. Classes are the templates of the objects we want to create. We first define a class, then produce objects from it.

2. Giving a name to an object is called assignment and we call this names variables in programming jargon. This is because although a name refers to only one object, the object it refers to can be changed any time.

   We need them to hold a data for later use in a program. The data might expected to be changing in different parts of a program so giving it a name would be easier for us instead of dealing with its value, directly.

3. Messages are classified as unary, binary and keyword messages. We can count `reverse` and `asUppercase` as examples of unary messages; count + and `*` as binary messages; count `at:put:` and `removeKey:` as keyword messages.

4. We can deal with natural numbers like `3`, `10`, `25`; real numbers like `3.5`, `8.2`, `0.4e5` or fractions like `4/7`, `8/13`, `25/138`.

5.

```
"answer_3_5.st"

| theNumber |

Transcript show: 'Please enter a number to get its cube: '.
theNumber := stdin nextLine.
```

```
Transcript show: 'The cube of ', theNumber, ' is ',
(theNumber asInteger squared * theNumber asInteger)
printString, '.'; cr.
```

```
Please enter a number to get its cube: 3
The cube of 3 is 27.
```

6.

```
"answer_3_6.st"

| theNumbers arithmeticAverage |

Transcript show: 'Please enter two numbers separated by
space to get their arithmetic average: '.
theNumbers := stdin nextLine tokenize: ' '.

arithmeticAverage := ((theNumbers at: 1) asInteger +
(theNumbers at: 2) asInteger) / 2.

Transcript show: 'The arithmetic average of ', (theNumbers
at: 1), ' and ',  (theNumbers at: 2), ' is ',
arithmeticAverage printString, '.'; cr.
```

```
Please enter two numbers separated by space to get their
arithmetic average: 10 20
The arithmetic average of 10 and 20 is 15.
```

7.

```
"answer_3_7.st"

| definitions theWord |

definitions := Dictionary new.

definitions at: 'programming language' put: 'We call the
command groups which provide a communication between us and
computer hardware as programming languages.'.
definitions at: 'source code' put: 'The code we wrote in a
programming language (not the result of a compilation or
interpretation process) is called source code.'.
definitions at: 'virtual machine' put: 'The converting
process from byte-code to machine code is done by programs
```

```
called virtual machines.'.
definitions at: 'cross-platform' put: 'Cross-platform is the
name for being able to run a software on different computer
architectures, like different operating systems or different
processors.'.

Transcript show: 'Please enter a word to get its definition:
'.

theWord := stdin nextLine.

Transcript show: theWord, ': ', (definitions at: theWord
asLowercase); cr.
```

```
Please enter a word to get its definition: programming
language
programming language: We call the command groups which
provide a communication between us and computer hardware as
programming languages.
```

We put an `asLowercase` message to the last expression because the keys are case sensitive and via this method, the program will be able to find the correct key even if the user enters, for example, *Programming Language*.

8. In `Array`s, we have a collection of data which is held in order and accessed by a numbered index. In `Dictionary`s, we also have a collection data but not held in any order. So, one of the reasons might be about the structure we want to use. For example, we can create a `Students` object as `Array` and reach the information of a student by a numbered index like `Student at: 1`. Or we can choose to keep students according to their name via a `Dictionary` class and reach a student's information by writing `Student at: Canol Gökel`.

   An other reason might about the performance. `Array`s keep their data consecutively inside the computer memory and the access time for an individual element is always the same while in `Dictionary`s the time increases as the element size increases. Though it is not a significant increase for not too big `Dictionary`s.

Chapter 4

1. The ultimate goal of control messages is to give the user opportunity to change the flow of the program instructions. Otherwise we won't be able to make decisions or execute a part of the program more than once.

There are two types of control messages: selective and repetitive. `ifTrue:` and `ifFalse:` are examples of selective control messages while `whileTrue:` and `to:do:` are examples of repetitive control messages.

2. Blocks are groups of expressions which we can pass to control messages. They are one of the things that make it possible to have control messages.

3. Definite loops are loops we know how many times it will repeat itself. The end of indefinite loops, however, depends on a certain condition to occur. `to:do:` is an example of definite loop messages while `whileTrue:` is an example of indefinite loops.

4.

```
"answer_4_4.st"

| theLength |

Transcript show: 'Please enter the length of the edge of the
diamond: '. t
heLength := stdin nextLine asInteger.

1 to: theLength do: [ :x |
    1 to: (theLength - x) do: [ :y |
        Transcript show: ' '.
    ].

    Transcript show: '/'.

    1 to: ((x - 1) * 2) do: [ :y |
        Transcript show: ' '.
    ].

    Transcript show: '\'; cr.
].

1 to: theLength do: [ :x |
    1 to: (x - 1) do: [ :y |
        Transcript show: ' '.
    ].

    Transcript show: '\'.
    1 to: ((theLength - x) * 2) do: [ :y |
        Transcript show: ' '.
    ].

    Transcript show: '/'; cr.
].
```

```
Please enter the length of the edge of the diamond: 5
    /\
   /  \
  /    \
 /      \
/        \
\        /
 \      /
  \    /
   \  /
    \/
```

5.

```
"answer_4_5.st"

| theNumber isPrime i |

Transcript show: 'Please enter a number: '.
theNumber := stdin nextLine asInteger.

i := 2.

[theNumber \\ i = 0] whileFalse: [
    i := i + 1.
].

(i = theNumber) ifTrue: [
    Transcript show: theNumber printString, ' is a prime
number.'; cr.
] ifFalse: [
    Transcript show: theNumber printString, ' is not a prime
number. It is devidable by ', i printString, '.'; cr.
].
```

```
Please enter a number: 3
3 is a prime number.
```

6.

```
"answer_4_6.st"

99 to: 1 by: -1 do: [ :x |
    Transcript show: x printString, ' bottles of beer on the
wall, ', x printString, ' bottles of beer.'; cr.

    (x > 1) ifTrue: [
        Transcript show: 'Take one down and pass it around,
', (x - 1) printString, ' bottles of beer on the wall.'; cr.

    ] ifFalse: [
        Transcript show: 'Take one down and pass it around,
no more bottles of beer on the wall.'; cr.
    ].

    Transcript cr.
].

Transcript show: 'No more bottles of beer on the wall, no
more bottles of beer.'; cr.
Transcript show: 'Go to the store and buy some more, 99
bottles of beer on the wall.'; cr.
```

7. With definite loop:

```
"answer_4_7_a.st"

1 to: 10 do: [:x | x printNl]
```

```
1
...
10
```

With indefinite loop:

```
"answer_4_7_b.st"

| x |

x := 1.

[x <= 10] whileTrue: [
    x printNl.
    x := x + 1.
]
```

```
1
...
10
```

First one feels the right one because we didn't have to declare a variable and didn't have to increment it manually. Also we didn't spend time thinking about the right conditional, we just wrote the range. This shows we should write a loop using definite loop techniques whenever the range of the loop is known before we enter into it.

8. It is impossible to write this program with definite loops because we don't know when the user is going to finish the program.[5] In other words, how many times the loop will repeat executing itself depends on the task user want to accomplish, which the program cannot know, so we (as the programmer) don't know the range of the loop before entering into it.

   By using indefinite loop:

```
"answer_4_8.st"

| newNumber howManyNumbers sum arithmeticAverage |

newNumber := 0.

howManyNumbers := 0.
sum := 0.
arithmeticAverage := 0.

[newNumber = 'finish'] whileFalse: [
    Transcript show: 'Arithmetic averages of the numbers so
far is: ', arithmeticAverage printString; cr.

    Transcript show: 'Please enter a new number or "finish"
to exit '.
    newNumber := stdin nextLine.

    (newNumber isNumeric) ifTrue: [
        sum := sum + newNumber asInteger.
```

5 Actually, it is possible but it is hard and unnecessary.

```
        howManyNumbers := howManyNumbers + 1.
        arithmeticAverage := sum / howManyNumbers.
    ].

    Transcript cr.
].
```

```
Arithmetic averages of the numbers so far is: 0
Please enter a new number or "finish" to exit: 2

Arithmetic averages of the numbers so far is: 2
Please enter a new number or "finish" to exit: 4

Arithmetic averages of the numbers so far is: 3
Please enter a new number or "finish" to exit: finish
```

9.

```
1 to: 1000 do: [:x | x printNl].
```

```
1
...
1000
1
```

Chapter 5

1. Polymorphism is the name of the concept to determine which method to execute according to the type of the class of the object we sent the message.

   Observing the attributes and behavior of the ancestor class when deriving a new class from it is called inheritance.

   Encapsulation is hiding the inner working details of an object from outside world.

2.

```
"answer_5_2.st"

Object subclass: Human [
    | name age |

    setName: aName [
        name := aName.
    ]
    getName [
        ^name
    ]

    setAge: anAge [
        age := anAge.
    ]

    getAge [
        ^age
    ]

    introduceYourself [
        Transcript show: 'Hello, my name is ', name, ' and
I''m ', age printString, ' years old.'; cr.
    ]

    > aHuman [
        ^age > aHuman getAge
    ]

    < aHuman [
        ^age < aHuman getAge
    ]

    = aHuman [
        ^age = aHuman getAge
    ]
]

Human subclass: Man [
    | money handsomeness |

    setMoney: amountOfMoney [
        "Amount of money out of 10"

        money := amountOfMoney.
    ]

    getMoney [
        ^money
    ]

    setHandsomeness: rateOfHandsomeness [
        "Handsomeness rate out of 10"

        handsomeness := rateOfHandsomeness.
    ]

    getHandsomeness [
        ^handsomeness
    ]

    > aMan [
        (self getName = 'Canol Gökel') ifTrue: [
            ^true
        ] ifFalse: [
            ^(self getMoney + self getHandsomeness) > (aMan
getMoney + aMan getHandsomeness)
        ]
    ]

    < aMan [
        (self getName = 'Canol Gökel') ifTrue: [
            ^false
        ] ifFalse: [
            ^(self getMoney + self getHandsomeness) < (aMan
getMoney + aMan getHandsomeness)
        ]
    ]
    = aMan [
        (self getName = 'Canol Gökel') ifTrue: [
            ^false
        ] ifFalse: [
            ^(self getMoney + self getHandsomeness) = (aMan
getMoney + aMan getHandsomeness)
        ]
    ]
]

Human subclass: Woman [
    | honesty generosity |
    setHonesty: rateOfHonesty [
        "Honesty rate out of 10"

        honesty := rateOfHonesty.
    ]

    getHonesty [
        ^honesty
    ]

    setGenerosity: rateOfGenerosity [
        "Generosity rate out of 10"

        generosity := rateOfGenerosity.
    ]

    getGenerosity [
        ^generosity
    ]

    > aWoman [
        ^(self getHonesty + self getGenerosity) > (aWoman
getHonesty + aWoman getGenerosity)
    ]

    < aWoman [
        ^(self getHonesty + self getGenerosity) < (aWoman
getHonesty + aWoman getGenerosity)
    ]

    = aWoman [
        ^(self getHonesty + self getGenerosity) = (aWoman
getHonesty + aWoman getGenerosity)
    ]
]

| man1 man2 man3 woman1 woman2 |

man1 := Man new.
man1 setName: 'Michael Cooper'.
man1 setAge: 32.
man1 setMoney: 9.
```

```
man1 setHandsomeness: 7.

man2 := Man new.
man2 setName: 'Paul Anderson'.
man2 setAge: 28.
man2 setMoney: 7.
man2 setHandsomeness: 8.

man3 := Man new.
man3 setName: 'Canol Gökel'.
man3 setAge: 24.
man3 setMoney: 1.
man3 setHandsomeness: 1.

woman1 := Woman new.
woman1 setName: 'Louise Stephney'.
woman1 setAge: 26.
woman1 setHonesty: 6.
woman1 setGenerosity: 7.

woman2 := Woman new.
woman2 setName: 'Maria Brooks'.
woman2 setAge: 32.
woman2 setHonesty: 8.
woman2 setGenerosity: 6.

(man1 > man2) printNl.
(man1 < man2) printNl.
(man3 > man1) printNl.
(woman1 > woman2) printNl.
(woman1 < woman2) printNl.
```

```
true
false
true
false
true
```

3.

```
"answer_5_3.st"

Number extend [
    cubed [
        ^self * self * self
    ]
]

3 cubed printNl.
```

```
27
```

4. These two keywords are used to refer to the receiver object of the message, inside the class definition. Whenever we send a message to `self` or `super` keyword, the interpreter sends the message to the current object in context.

   The difference of `super` keyword from `self` is that when we send a message to `super`, the search of the method begins from the superclass of the receiver object. So the method inside the superclass is invoked instead of the method of the receiver object's class.

Alphabetical Index

Postface

Now, we are at the end of the book, our survey of learning the basics of computer programming using GNU Smalltalk. We hope you enjoyed this book. You can always send feedback to me via the email address in the section below.

About the Author

Canol Gökel is a student at Hacettepe University, Department of Electrical and Electronics Engineering, Turkey. He is also a member of Hacettepe Robotics Society. He likes programming computers, microcontrollers and also likes learning (not using :P) new programming languages. You can reach him via the email address: canol@canol.info