



Object Oriented Program Development

Undergraduate Programming CourseBook

Dr. Le, Minh Duc

© 2021, 2023

Contents

List of Figures	ii
List of Tables	iii
Preface	1
1 Overview of Programming Languages	5
1.1 Programming Language Landscape: An Overview	5
1.1.1 Why So Many Languages?	6
1.1.2 Language Classification	6
1.2 Programming Language Features	11
1.2.1 Expressive power	11
1.2.2 Ease of use	11
1.2.3 Ease of implementation	11
1.2.4 Standardisation	12
1.2.5 Open source	12
1.2.6 Language tool quality	12
1.2.7 User base	13
1.3 What Makes a Good Programming Language?	13
1.4 Why Study Programming Languages?	14
Chapter 1 Question	14
Chapter 1 Exercise	15
2 The Java Programming Language	20
2.1 Motivation	20
2.2 Language Syntax and Semantics	21
2.2.1 Syntax	21
2.2.2 Syntax Grammar	22
2.2.3 Syntax Tree	25
2.2.4 Nested box diagram – An Alternative Representation of Syntax Tree	26
2.2.5 Semantics	27
2.3 The Java Language Syntax	28
2.3.1 Lexical grammar	28

2.3.2	Syntactic grammar	29
2.3.3	Additional Java grammar notation	30
2.4	Identifier	30
2.5	Method Declaration	31
2.6	Blocks and Statements	32
2.6.1	Block (§14.2)	33
2.6.2	Statement	35
2.6.3	if Statement	36
2.6.4	while Statement	37
2.6.5	for Statement	37
2.6.6	switch Statement	39
2.7	Expression	40
2.7.1	Primary Expression	41
2.7.2	Literal	42
2.7.3	Method Invocation	42
2.7.4	Array Access	43
	Chapter 2 Exercise	45
3	Introduction to Compiler and Virtual Machine	48
3.1	Compiler	48
3.1.1	What Is Compiler?	48
3.1.2	Compilation Levels	49
3.1.3	Compiler Operations	50
3.1.4	Lexical and Syntactic Analysis	51
3.1.5	Semantic Analysis	52
3.1.6	Target Code Generation	54
3.1.7	Code Improvement	55
3.2	Virtual Machine	55
3.3	Java Virtual Machine	56
3.3.1	Storage Management	57
3.3.2	Class File Format	58
3.4	Java Program Execution	60
	Chapter 3 Exercise	62
4	Verifiable Program Development	66
4.1	Basic Terminology	67

4.1.1	Procedure	67
4.1.2	Behaviour and Implementation	68
4.1.3	Types of Procedure	70
4.1.4	Procedural Abstraction	71
4.1.5	Procedure Invocation	73
4.2	Verifiable Procedure Design	73
4.3	Design Specification Language	74
4.3.1	Block Comment Format	75
4.3.2	Behaviour Specification Structure	75
4.3.3	Language Constructs	77
4.3.4	What Makes a Good Specification?	78
4.4	Verifiable Program Design	80
4.4.1	Preliminaries	80
4.4.2	Structured Program Design	84
4.4.3	Specifying the Program Class	85
4.4.4	Specifying Procedure <code>main</code>	85
4.4.5	Specifying Functional Procedures	86
4.5	Implementation in Java	87
4.6	Application Example: Coffee Tin Game	90
4.6.1	Problem definition	90
4.6.2	Analysis	91
4.6.3	Design	92
4.6.4	Code	97
Chapter 4 Exercise	103
5	Introduction to Object Oriented Program	105
5.1	Motivating Example	105
5.2	Why Object Oriented Program?	106
5.2.1	Example: Greeting Conversation 2	107
5.2.2	Limitations of Procedural Program	107
5.2.3	How Does Object Oriented Program Help?	108
5.2.4	Historical Context	109
5.3	What Is OOP?	110
5.4	Object and Class	113
5.5	Class Design Diagram	115
5.6	Information Hiding	116

5.7	Comparing OOP with Procedural Program	116
5.8	Benefits of OOP	117
5.9	OOP Development Process	117
5.9.1	Example: Floating buoys management software	117
5.9.2	Overview of the Development Method	118
5.9.3	Create a function design model	119
5.9.4	Identify objects and attributes	121
5.9.5	Identify self and required behaviour	122
5.9.6	Define classes	123
5.9.7	Identify associations	124
5.9.8	Class diagram	125
	Chapter 5 Exercise	127
6	Object Oriented Design Fundamentals	128
6.1	Motivation	129
6.2	Design Terms	130
6.2.1	Data abstraction	131
6.2.2	Abstract concept	131
6.2.3	Class	131
6.2.4	Class vs. abstract data type	132
6.2.5	Object	132
6.3	Design Method	133
6.3.1	Method Overview	134
6.4	Design Specification Language	135
6.4.1	Design Notation	135
6.4.2	Text-Based Set Notation	136
6.4.3	Useful Object Data Types	136
6.5	Header Specification: Specifying the Abstract Concept	140
6.5.1	Overview	140
6.5.2	Choosing a Name	140
6.5.3	Writing the Overview	141
6.5.4	Defining Attributes	141
6.5.5	Defining Abstract Object	143
6.5.6	Defining Abstract Properties	144
6.6	Specifying Concrete Attribute Types	147
6.7	Specifying the State Space	149

6.7.1	Defining instance variables	150
6.7.2	Annotating instance variables with domain constraints	151
6.8	Essential Design Annotations	154
6.9	Specifying the Behaviour Space	155
6.9.1	Essential Operation Types	155
6.9.2	How to Choose the Operations?	155
6.9.3	General Specification Guidelines	156
6.9.4	Annotation Guidelines	157
6.9.5	Constructor	159
6.9.6	Mutator	161
6.9.7	Observer	162
6.9.8	Default	163
6.9.9	Helper	164
6.10	Specifying the Collection Classes	167
6.10.1	Collection Class Marker	168
6.10.2	Using DOpt with Specialised Types	168
6.10.3	Constructor	169
6.10.4	Mutator	169
6.10.5	Observer	170
Chapter 6 Exercise	171
Appendix 6.A	Customer	174
Appendix 6.B	IntSet	176
7	Design Review and Coding	179
7.1	Reviewing the Design	179
7.1.1	Check Header Specification	180
7.1.2	Check State Space Specification	180
7.1.3	Check Behaviour Space Specification	180
7.2	OOPChecker: A Design Validation Tool	181
7.2.1	Overview	181
7.2.2	Essential Design Rules	182
7.2.3	Set up and usage	183
7.3	General Implementation Guidelines	183
7.4	Constructor	184
7.5	Mutator	185
7.6	Observer	187

7.7	Default Operation	189
7.7.1	Operation <code>toString</code>	189
7.7.2	Operation <code>equals</code>	190
7.7.3	Operation <code>hashCode</code>	192
7.8	Helper	192
7.8.1	Data validation	192
7.8.2	Operation <code>repOK</code>	194
7.8.3	Utility	195
7.9	Implementing the main method	196
7.9.1	A Basic Customer Relationship Manager (CRM)	196
7.9.2	Integers	198
7.9.3	Working with Wrapper Classes	198
	Chapter 7 Exercise	200
	Appendix 7.A Customer	201
	Appendix 7.B IntSet	204
8	Design Issues	209
8.1	Basic Design Issues	209
8.1.1	Information Hiding	209
8.1.2	Encapsulation	210
8.1.3	Separation of Concerns	212
8.1.4	Operation Overloading	212
8.2	Example: Customer	214
8.3	Object Life Cycle	214
8.3.1	Object Initialisation	215
8.3.2	Object Reference	216
8.3.3	Stack and Heap	217
8.3.4	Passing An Object as Argument	220
8.4	Private Constructor	222
8.5	Object Cloning	223
8.6	Inner Class	225
8.7	Generic Class	227
	Chapter 8 Exercise	229
9	Introduction to Design Patterns	233
9.1	What Is Design Pattern?	233

9.2	Pattern Definition Format	234
9.3	Class Design Patterns	235
9.4	Singleton	235
9.4.1	Problem	235
9.4.2	Solution	236
9.4.3	Example	237
9.4.4	Consequences	239
9.5	Factory Method	239
9.5.1	Problem	239
9.5.2	Solution	240
9.5.3	Example	241
9.5.4	Consequences	242
9.6	Derived Attribute	242
9.6.1	Problem	243
9.6.2	Solution	243
9.6.3	Example	244
9.6.4	Consequences	246
9.7	Recursive Class	246
9.7.1	Background: Recursive Definition	247
9.7.2	Problem	247
9.7.3	Solution	248
9.7.4	Example	250
9.7.5	Consequences	253
	Chapter 9 Exercise	254

10 Abstract Data Type:

	Object Oriented Design and Implementation	255
10.1	Overview	256
10.1.1	Abstract Data Type	256
10.1.2	List and Tree	256
10.1.3	Java Support	257
10.2	Overall Design Approach	258
10.3	Tree	258
10.3.1	Abstract Concept	259
10.3.2	Operations	264
10.3.3	Design Approach	266

10.3.4 Bottom-Up Design	268
10.3.5 Top-Down Design	271
10.3.6 Implementation	274
10.4 List	283
10.4.1 Abstract Concept	283
10.4.2 Operations	284
10.4.3 Design Approach	286
10.4.4 ‘Pure’ Recursive List	286
10.4.5 Linked List	292
10.4.6 Implementation	296
Chapter 10 Exercise	317
Appendix 10.A Node	319
Appendix 10.B Edge	320
Bibliography	324

List of Figures

1.1	Programming language spectrum (Adapted from [28]).	6
1.2	Von neumann architecture (Adapted from [29]).	9
1.3	Structuring a Java project in Eclipse.	15
2.1	Syntax analysis of the example text.	21
2.2	Syntax analysis of the Java program Gcd in Listing 2.1.	22
2.3	The context-free grammar rule for variable declaration in Java.	24
2.4	An example syntax tree.	26
2.5	An example NBD.	27
2.6	Method declaration example.	32
2.7	Method header example.	32
2.8	Method declaration with exception.	33
2.9	Local variable declaration example.	35
2.10	If statement example.	36
2.11	The while statement example.	37
2.12	The for statement example.	38
2.13	Enhanced for statement example.	39
2.14	The switch statement example.	40
2.15	Method invocation without arguments.	43
2.16	Method invocation with arguments.	43
2.17	Array access example.	44
3.1	Overview of compiler (Adapted from [28]).	49
3.2	Compilation levels (Adapted from [28]).	49
3.3	Compiler operations (Source: [28]).	51
3.4	Partial syntactic analysis for the while and putint statements in Gcd (Adapted from [28]).	53
3.5	Semantic analysis of program Gcd (Adapted from [28]).	54
3.6	Partial assembly code for program Gcd (Adapted from [28]).	55
3.7	Java VM and platform.	56
3.8	Java program development: (1) compilation, (2) execution.	57
3.9	Program execution procedure.	61
4.1	Procedural abstraction types of the procedure swap.	72

4.2	Applying the specification criteria to the procedure swap.	80
4.3	The structure of program Sum.	82
4.4	The structure of program Sum with multiple classes.	83
4.5	Structure of program CoffeeTinGame.	93
5.1	Overcoming Problem 1 and Problem 2 with OOP.	109
5.2	Overcoming Problem 3 with OOP.	109
5.3	The design diagram of class Person.	115
5.4	The activity diagram of the floating buoys management software.	120
5.5	Class diagram of the buoy program.	126
6.1	A bare design of class Person.	129
6.2	Design term map.	130
6.3	Class and object design diagram.	135
6.4	Choosing the class name for Customer and IntSet.	140
6.5	Writing <code>@overview</code> in a note box and attaching it to the class.	141
6.6	Adding attributes to the class design.	142
6.7	Abstract objects of Customer, IntSet and Integer.	144
6.8	Customer with concrete types.	148
6.9	IntSet with concrete type.	149
6.10	Customer's representation.	150
6.11	IntSet's representation.	151
6.12	Customer representation with constraints.	152
6.13	IntSet representation with constraints.	153
6.14	The design annotations.	154
6.15	Customer operations.	156
6.16	IntSet operations.	156
6.17	An initial design diagram for the greeting conversation program.	172
7.1	OOPCHECKER as an Eclipse plugin.	182
8.1	An extended design of class Customer.	214
8.2	The object life cycle showing the high-level states and the transitions between them.	215
8.3	Example object reference created for a Customer object.	217
8.4	Two snapshots of the stack and heap diagram.	218
8.5	The stack and heap diagram of CustomerAppSimple.	220
8.6	A partial stack and heap diagram of CustomerApp.	222

8.7	The design of inner class <i>Customer2.Address</i>	226
8.8	The design of generic class <i>Set<T></i>	228
9.1	Design diagram of singleton pattern.	236
9.2	Singleton class <i>CustomerMan</i>	237
9.3	Design diagram of factory pattern.	240
9.4	Factory class <i>CustomerFactory</i>	241
9.5	Design diagram of derived attribute pattern.	244
9.6	Class <i>Customer</i> with derived attribute <i>age</i>	245
9.7	Design diagram of recursive class.	249
9.8	The design of recursive class <i>Fact</i>	250
10.1	An illustration of the list and tree abstractions of a document.	255
10.2	A binary-tree representation of the list (2,3,5,7).	257
10.3	Tree example.	259
10.4	Constructing the tree in Figure 10.3 from the bottom up.	262
10.5	Constructing the tree in Figure 10.3 from the top down.	263
10.6	A rooted tree with 12 nodes.	265
10.7	Class <i>Node</i>	267
10.8	Class <i>Edge</i>	267
10.9	The recursive, bottom-up design of class <i>Tree</i>	268
10.10	The recursive, top-down design of class <i>Tree</i>	272
10.11	The ‘pure’ recursive design of <i>List</i>	287
10.12	Cell-based design for list (Adapted from [1]).	292
10.13	The linked ‘recursive’ design of <i>List</i>	293

List of Tables

4.1	Logical notation	79
4.2	Design specification quality criteria	79
5.1	Object definitions	122
5.2	Self behaviour	122
5.3	Required behaviour	123
5.4	Class definitions	124
5.5	Associations	125
6.1	Design constraint table	136
6.2	Set notation	136
6.3	Wrapper classes.	137
6.4	Customer's domain constraints.	145
6.5	IntSet's domain constraints.	146
7.1	The essential design rules set that are supported by OOPCHECKER.	182
8.1	Default attribute values.	216

Preface

Object oriented program development (OOPD) primarily concerns the effective use of an object oriented programming language in designing and coding computer programs. The core elements of such object oriented programs (OOPs) are objects and the interactions between them that together define the program's behaviour. Objects and their interactions need to closely model the data and behaviour of the real-world entities that exist in the problem domain. The aim of OOPD is to effectively identify and develop in an OOPL all the relevant object types, called *classes*, and make these available for use as templates for creating objects. The design specification and code of each class are the determinants of its quality: the code needs to be correct with regards to the design specification. A well-crafted design specification plays a crucial role in verifying the code's correctness. A program that has such a specification is a *verifiable program*. A main objective of OOPD is to ensure that each constructed program is a verifiable one.

Learning OOPD requires understanding the OOPL's features and a development method that effectively leverages these features to produce good quality classes. These are also the two main topics of this book. In the first topic, the book takes readers through a brief tour of the programming language landscape, highlighting the common language features and the key language enabling tools. In the second topic, the book discusses an adapted OOPD method, which has been extended with some of the author's own ideas. The original method was presented in an authoritative book by Liskov and Guttag [20]. In the current book, the author had updated this method to use the latest OOPL features, the most noticeable of which is *annotation*. In principle, annotation is used to express the essential class design rules directly in the program text. It is argued that such embedded design rules help improve the program's quality. There are two main reasons for this. First, the design annotations form the design specification that helps the client know exactly what to expect. Second, since annotation is a language construct, the design rules can be parsed and fed into a tool to validate the program's behaviour. This book is accompanied by a tool named OOPCHECKER, which is made publicly available for students to download and use with the development method.

To ease the application of the method in practice, the book discusses a number of core design issues and patterns. Among these, the recursive class design pattern, which is based on the fundamental design concept of recursion, is a contribution of the book. These design patterns are proved useful in building the object oriented designs of two important abstract data types (list and tree).

Unlike other books in OOPD, which tend to broadly cover class design together with all the object oriented principles (including inheritance and polymorphism), this book believes that objects and classes should be the core issues to be discussed in detail before introducing other principles. The major emphasis of this book on verifiable program design specification reflects the author's belief that a good program is a well-thought-out program. Consequently, a good programmer is a programmer that has a design thinking and understands its close relationship with a target programming language platform. This book hopes to demonstrate that there is much to be discovered in the centre of the 'galaxy' of objects and classes!

Goals

This book is suitable for intermediate to advanced programming courses whose main emphasis is on OOPD. The aim of the book is to systematically present the essential concepts, techniques and tools for developing verifiable programs.

The book has three main objectives. The first objective is to introduce the programming language concept and a classification of languages. This will also include the basic principles of the modern compiler – a key language tool that processes programs written in a language. The second objective is to explain the concept of verifiable program and how to develop such a program. This is a generic concept that covers both procedural and object oriented programs. The third objective is to discuss the transition from procedural to object oriented programming and describe how to develop a basic OOP. A key component of this is an annotation-based class design specification method, which helps develop the verifiable classes of an OOP.

Prerequisites

To make the most of this book, students should ideally have already completed the following courses (or equivalent):

- Introduction to programming (preferably) in Java.
- Discrete mathematics (in particular, the fundamentals of logic)

Approach

A core component of a verifiable program is verifiable procedure, whose behaviour is well defined. Verifiable procedures exist in two forms: stand-alone and object.

Verifiable *stand-alone procedure* is arguably a perfect starting point for learning how to write good behaviour description, independent of any concerns with object oriented design. This book, thus, takes the approach by Liskov and Guttag [20] to first explain the concept of verifiable program in the context of procedural design. Once this concept has been grasped, the book introduces OOP, highlighting the key transitional concepts to be learnt. Among these is the adaptation of the verifiable procedure concept to define verifiable *object procedure* (a.k.a *operation*). This then becomes a basis to define the development method for verifiable OOP.

The book next discusses the fundamental class design issues and apply these to, more broadly, define a number of basic class design patterns. Although the design patterns introduced in this book are at the introductory level, they help lay a foundation for learning more sophisticated patterns at a later stage. To demonstrate the practicality of the development method and design patterns, the book applies these to develop object oriented solutions for two key abstract data types (list and tree).

Java is chosen as the programming language used in the book because it is a popular platform-neutral object oriented programming language. However, we believe that the concepts and methods discussed in this book can be applied to other OOPLs (e.g. C#) that support similar language features.

Structure of the Book

This book is organised into 10 chapters:

- *Chapter 1*: gives a brief guided tour of the programming language landscape, highlighting the common language features and the role and features of OOPL.
- *Chapter 2*: explains the syntax and semantics of the Java language.
- *Chapter 3*: introduces the operational principles of compiler and virtual machine.
- *Chapter 4*: presents the concept of verifiable program and how to develop such a program.
- *Chapter 5*: introduces OOPD and compare and contrast OOP with procedural program.
- *Chapter 6*: discusses the design component of the development method for OOP.
- *Chapter 7*: describes the coding component of the development method for OOP.
- *Chapter 8*: presents the fundamental class design issues.
- *Chapter 9*: introduces the concept of design pattern and defines a number of basic class design patterns.
- *Chapter 10*: discusses object oriented solutions for two fundamental abstract data

types (list and tree).

Exercises

To help practise the concepts introduced in the book, each book chapter contains a set of exercises. The book is also accompanied by an instructor workbook that contains answers to and instruction guides for the exercises.

Notation

The essential notational style used in the book is described below:

- Technical concepts and Java code are written in **fixed font**. For example, class `Student` refers to the Java class named “`Student`”.
- The singular and plural forms of a technical concept may be used to refer to instance(s) (i.e. object(s)) of this concept. For example, the phrase “a `Student` named Le Van Anh” refers to a specific object of class `Student`, while the plural form `Students` refers more generally to the objects of this class.
- Names of software frameworks and software are written in **ROMAN ALL CAPS FONT**.

Chapter 1

Overview of Programming Languages

Objectives

- ✓ Understand the programming language landscape and the spectrum of programming languages
- ✓ Understand the common programming language features
- ✓ Explain the underlying principle that makes a good programming language
- ✓ Understand the motivation for studying programming language

A programmer communicates to a computer through a programming language is analogous to a person who communicates in a natural language. The more she knows the language, the better she can communicate programs or ideas in that language. Indeed, the level of knowledge of the programming language is a key factor that distinguishes between a good and a great programmer.

Getting to know a language and using it fluently takes time and effort. There are many programming languages currently out there and new ones are constantly being created. Languages are designed either to solve a new set of problems or to solve the same problems in a different (preferably better) way. Thus, programmers usually need to know more than one languages, so that she can choose the most suitable one for the task at hand. This chapter gives a brief tour of the programming language landscape. It attempts to highlight the most common features and what makes a good language. It also explains why it is important for programmers to take time and effort to learn the languages that they intend to use.

1.1 Programming Language Landscape: An Overview

Although the exact number of programming languages (PLs) in existence is unknown, it has been well reported that there are thousands of PLs [28]. A keyword search for “programming language survey” on Google scholar¹ for papers in the last five years return survey papers for specific types of languages (e.g. [3, 19, 25, 32]). However, there are no papers in the result about all the PLs. A recent developers survey by Stack-

¹<https://scholar.google.com>

Overflow² lists the 25 most popular PLs among the developers. This list is quite diverse, covering the two major types of languages: declarative and imperative.

This section gives an overview of the programming language landscape by first explaining why there is a such a high number of languages. It then introduces a language classification, which organises the languages based on their computational model.

1.1.1 Why So Many Languages?

Scott [28] suggests the three reasons for why there are so many PLs: evolution, special purposes and personal preference. First, PLs do not stay the same but change over time to find ways of solving new problems and/or to find better ways of solving the existing ones. Scott [28] states two important evolutionary steps that occurred in the PL history. The first step involved replacing the goto statement, which was commonly used to regulate the program flow in the early procedural PLs (such as Fortran, Cobol and Basic), by the loop and case statements. The second evolutionary step occurred around a decade after, when block-based PLs (such as Agol, Pascal and Ada) were losing ground to object oriented PLs (such as SmallTalk and C++).

Second, new languages are constantly being invented to solve problems in certain specific domains. These domains tend to have special requirements that do not fit the capabilities of the existing PLs. This calls for the development of new PLs to address them. Third, PL, like natural language, can be personal. Among a group of alternative PLs (e.g. Java and C#), some developers would prefer one language over the other. This makes it difficult for developers to come to an agreement about a ‘universal’ PL for all.

1.1.2 Language Classification

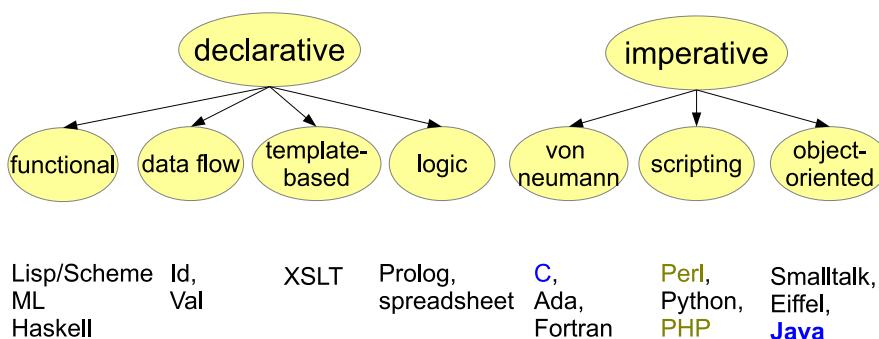


Figure 1.1: Programming language spectrum (Adapted from [28]).

²<https://insights.stackoverflow.com/survey/2020>

The classification shown in Figure 1.1 is based on the model of computation. In principle, there are two major types of PLs: *declarative* and *imperative*. It is important note that these two top-level categories are not necessarily disjoint. With recent PL developments, they should be more correctly interpreted as describing two features set of a PL. Indeed, modern languages can have features that come from both both sets. For example, although Java is fundamentally an object-oriented PL, starting from version 1.8, it supports functional programming through lambda expression.

Declarative PL

In principle, a declarative PL focuses on expressing a solution in terms of *what* steps need to be taken. The PL platform is responsible for converting those into the detailed instructions, suitable for being performed by the computer. The figure list five sub-types of declarative PL, which include functional, data flow, template-based and logic. A *functional PL* is one whose considers function as the main construct and provides mechanism for writing programs recursively in terms of functions. A program is a top-level function which is incrementally refined in terms of smaller ones, and each of these is further refined in terms of even smaller ones. This process stops when all the functions are those provided by the PL. Examples of functional PL are Lisp, ML and Haskell [28].

A *data flow PL* treats a program as flows of data that pass through processing nodes. Each node is triggered by an input flow of data and its output (if any) would be sent to the next node(s). The nodes can potentially execute in parallel. Examples of data flow PL include Id, Sisal (which evolves from Id) and Val [28].

A *logic PL* has its root in predicate logic. It considers a program as a set of logic statements, describing relationships among the domain terms, and attempts to determine (search for) values that satisfy the program. Examples of logic PL include Prolog, Structured Query Language (SQL) and the XSLT scripting language [28].

Example 1.1 Declarative PL

Listing 1.1, which was adapted from a wiki page of the language³, lists the definition of the function named gcd in Haskell. This function takes takes as input two integers a and b and computes their greatest common divisor⁴. The function definition uses the fast Euclidean method⁵ of computing the GCD. This method uses a fact that GCD also

³https://wiki.haskell.org/99_questions/Solutions/32

⁴<https://mathworld.wolfram.com/GreatestCommonDivisor.html>

⁵<https://mathworld.wolfram.com/EuclideanAlgorithm.html>

divides the smaller number of the two input (b in this case) and the remainder (mod) of the division $\frac{a}{b}$.

The first line lists the type signature declaration of the function, which takes two integer inputs returns one integer output. The next three lines show the function definition which consists of two cases. The first case is applied when $b = 0$. In this case, we cannot divide a by b but can easily see that result is $\text{abs } a$ (abs means to take the absolute value). The second case is applied when $b \neq 0$. In this case, $\text{GCD}(a, b)$ is recursively defined in terms of $\text{GCD}(b, a \bmod b)$.

Listing 1.1: Program GCD in Haskell

```

1 gcd :: Integer -> Integer -> Integer
2 gcd a b
3     | b == 0      = abs a
4     | otherwise = myGCD b (a `mod` b)

```

Listing 1.2 shows the same program written in the Prolog language. The program consists in three propositions that establish the truth about $\text{gcd}(a, b, g)$. The first two terms (a, b) are input while the third term (g) is the GCD to be determined. The first proposition means that $\text{gcd}(a, b, g) = \text{true}$ if $a = b = g$ (same number). The second proposition means that $\text{gcd}(a, b, g) = \text{true}$ if $a > b$ and $\text{gcd}(c, b, g) = \text{true}$, for $c = a - b$. The third proposition states $\text{gcd}(a, b, g) = \text{true}$ if $a < b$ and $\text{gcd}(c, a, g) = \text{true}$ for $c = b - a$.

Listing 1.2: Program GCD in Prolog

```

1 gcd(A,B,G) :- A = B, G = A.
2 gcd(A,B,G) :- A > B, C is A-B, gcd(C,B,G).
3 gcd(A,B,G) :- B > A, C is B-A, gcd(C,A,G).

```

Although the logic rules of Prolog are different from Haskell's functional ones, it is clear from both programs that they consist in only high-level terms and their relationships and do not concern how the these are processed by the PL platform. In particular, we see no presence of variable modification in both programs. Internally, the PL platform would need to use some kind of algorithm which manipulate values in order to determine the GCD. However, these details are hidden from the programmer. This is a stark contrast to imperative programs, which we will discuss below. \square

Imperative PL

An imperative PL describes a solution in terms of *how* to perform a task. Figure 1.1 lists three sub-types of imperative PL, including von neumann, scripting and object-

oriented. *Von neumann* is the most successful type of language, which includes popular PLs of the past (such as as Fortran, Ada 83 and C) and continues to influence many popular PLs of the presence (such as Java and C#). A von neumann PL is designed to directly mimics the Von Neumann computer architecture. At the abstract level (see Figure 1.2), this architecture considers the computer as consisting three abstract components (the processing unit (CPU), run-time memory and I/O devices) and information exchange mechanisms between these components. What influences Von neumann PL is the relationship between CPU and memory. CPU holds the basic instructions that are performed on the data stored in memory. A key design feature of von neumann PL is the declaration of variables (memory storage of data) and a systematic modification of these variables (using various types of statement) with the intention to achieve the intended program outcome.

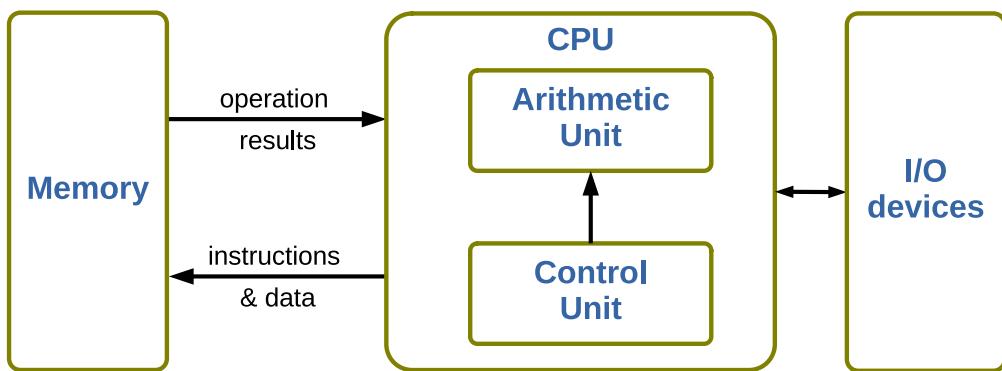


Figure 1.2: Von neumann architecture (Adapted from [29]).

According to recent surveys (such as that performed by StackOverflow²), scripting languages are gaining popularity and, in some cases, overtaking traditional general-purpose languages in programmer preference. A *scripting PL* is a type of Von neumann PL that is designed to speed up software development through the use of high-level expressions over a rich library of pre-defined components. Scripting PLs traditionally target specific problems (e.g. csh and bash for system task control). More recently, they are developed to address a wider range of problems. As of this writing, popular scripting PLs, such as TypeScript/JavaScript, are used for developing web applications. JavaScript, in particular, can be used not only to develop web pages (the front-end) but also to perform data processing on the server (the back-end).

Last but not least, *object-oriented PLs* (OOPLs) are closely related to the von Neumann counterparts in the assumption of the underlying architecture. However, OOPLs support a higher-level form of abstraction than functions and procedures. They consider programs as a set of objects interacting with each other to achieve the intended behaviour. An object has both state (the data) and operations (procedures) that enable

the observation and manipulation of the state over time. This programming model closely resembles how objects behave in the real-world and so it is the main reason why OOPLs (such as Java and C#) are one of the most successful type of PL. Some OOPLs (e.g. SmallTalk) are pure in that they do not mix in features from other sub categories shown in Figure 1.1. More modern OOPLs (such Java (version ≥ 1.8) and C#) include other features, such as functional programming with lambda expression. With its support for JavaScript, Java even incorporates some scripting language features.

Example 1.2 Imperative PL

Listings 1.3 and 1.4 list two versions of the program GCD in C and Java. Both programs assume that the input are non-negative. The two programs look remarkably similar in syntax, due primarily to a fact that the Java's syntax was deliberately designed to be similar to C's. We chose this example from [28] to demonstrate the nature of imperative PL. It is clear from both listings that the program's behaviour is realised by imperative statements (loop and conditional) that systematically manipulate values of the two input arguments.

Listing 1.3: Program GCD in C

```

1 int gcd(int a, int b) {           // C
2     while (a != b) {
3         if (a > b)
4             a = a - b;
5         else
6             b = b - a;
7     }
8     return a;
9 }
```

Listing 1.4: Program GCD in Java

```

1 static int gcd(int a, int b) {    // Java
2     while (a != b) {
3         if (a > b)
4             a = a - b;
5         else
6             b = b - a;
7     }
8     return a;
9 }
```



1.2 Programming Language Features

Given that there are many PLs in practice, a natural question to ask is whether or not there exists a common ground among them. If so, we can use that as basis to study and to compare and contrast between them. We argue that the set of features that characterise successful PLs suggested by Scott [28] can be adapted to characterise PLs in general. The common features among PLs are: expressive power, ease of use, ease of implementation, standardisation, open source, language tool quality and user base.

1.2.1 Expressive power

This is the extent to which a PL enables the construction of clear, concise and maintainable program code. A key measure here is the level of abstraction supported. In principle, the higher the abstraction, the more expressive the language. For example, we discussed in Section 1.1.2 how functional and logic PLs, which operate at a higher level of abstraction than the imperative counterparts, enable the program designer to think more abstractly about the solution.

1.2.2 Ease of use

This is the extent to which the language can easily be learned by its target users. Programmers, especially the beginners, would more easily learn how to use a PL if its syntax is compact and easy to understand. Successful PLs of the past, e.g. Basic and Pascal, were popular because they were easy to learn by different user groups. One way of achieving syntax ease-of-use is to learn from those successful past PLs and improve. A good example of this is the Java language. It uses the best syntactic features from two popular languages (C and C++) and combines them with object oriented features that were adapted from Eiffel and SmallTalk.

1.2.3 Ease of implementation

This is the extent to which the language can easily be implemented in its target computer platforms. These include general-purpose and special-purpose (e.g. embedded and mobile) platforms. This feature includes both the extent to which the PL specification is accessible to the platform vendors and the characteristics of the PL design that allow it to be implemented with ease in the target platforms. Examples of PLs that have enjoyed

significant successes with this feature are Basic, Pascal and Java. Java is a good, more recent example. First, the Java language specification is made freely available online⁶ for platform vendors to implement. Second, Java is designed to be platform neutral, which means that the same Java program can be executed on different platforms.

1.2.4 Standardisation

This is the extent to which the PL specification is standardised. A PL specification is a formal description of its syntax and semantics. Because a PL can potentially be implemented by a diverse community of vendors, its specification needs to be standardised so that all the implementations can process programs written in the language the same way. This is needed to ensure the code portability across platforms. It is also very important that the PL specification is frequently revised and updated as the language evolves so that vendors can update their implementations accordingly. A good example of standardisation can be found in the case of the Java language specification mentioned above. The specification is made available online and frequently updated to ensure its currency.

1.2.5 Open source

This is the extent to which a PL is equipped with an open source compiler. Having an open source compiler project helps grow the interest in the language and through that increases its user base. Further, an open source compiler helps ease the development burden of the PL inventor(s) because such a compiler is mostly developed and maintained by the community. In the Java case, the PL has an open source compiler for the standard-edition version, named OpenJDK⁷, which is an alternative to the official compiler provided by Oracle.

1.2.6 Language tool quality

This is the extent to which the language tool set (including the compiler and other supporting tools) helps not only generate good quality compiled code but ease developers in the program development process. Scott [28] gives two examples of successful PLs that have particularly good tool set. Fortran has a good compiler that generates efficient code, while Common Lisp⁸ provides tools that help programmers manage large projects.

⁶<https://docs.oracle.com/javase/specs/>

⁷<http://openjdk.java.net/>

⁸<https://common-lisp.net/>

The Java's rich tool set⁹ includes a variety of tools. Among these are the compiler (javac), java application launcher (java), debugger tool (jdb) and API documentation generator tool (javadoc).

1.2.7 User base

This is the extent of the user base of a PL. This includes sponsors and the programmer communities that have adopted the language. Having the support of powerful sponsors is important to ensure the continuity of resource stream for the language development and maintenance efforts. For example, two very successful OOPLs (e.g. Java and C#) are able to maintain their longevities because of the strong supports that they receive from their sponsors (e.g. Oracle (for Java) and Microsoft (for C#)). On the other hand, the user community plays a crucial role in the adoption of the language by programmers. There is no question that the larger the community, the more popular a PL. Examples of PLs that have been enjoying successes even without strong sponsor supports are Python¹⁰ and PHP¹¹.

1.3 What Makes a Good Programming Language?

Donald E. Knuth – regarded as the farther of algorithm analysis – summarises the essence of programming as “... explaining to human beings what we want a computer to do” [16]. This quote suggests an important, and often neglected, aspect of PL. That it is a communication medium between the the PL inventor and its PL users (the programmer and PL implementor). From this perspective, a good PL needs to balance between the desires of its stakeholders. On the one hand, the programmer perceives PL as a means of expressing algorithms for solving problems. Their main desire is for the PL to have these two features: expressive power and ease of use (see Section 1.2). On the other hand, the PL implementor views a PL as a mean to write instructions to be executed in a computing platform. Their main desire is thus ease of implementation and execution efficiency.

Reconciling these two conflicting groups of features is not easy. It is often the case that more expressive power requires more computing power and less efficient code. For example, declarative PLs are usually slower in performance, compared to

⁹<https://docs.oracle.com/javase/8/docs/technotes/tools/index.html>

¹⁰<https://www.python.org/>

¹¹<https://www.php.net/>

the imperative counterparts [28]. Making the right trade-offs is not exact science and requires experience and lots of experimentation. This also explains why there is no such thing as a perfect PL and that most PLs evolve towards achieving a better compromise over time.

1.4 Why Study Programming Languages?

Studying a PL involves understanding its purpose and learning the language specification enough to solve problems. The latter takes time and effort, especially when the language constantly evolves over time. The deeper and broader the knowledge about a language that a programmer has, the more productive she becomes at working with that language. Among the main benefits of studying a PL are:

- knowing the special features that can help write programs faster in certain contexts
- selecting language constructs suitable for a given situation
- using the language tool set to speed up the development process
- applying the best practices in one language to enhance another
- leveraging the knowledge of language technology (e.g. structured file parsing) to write better programs

Chapter 1 Question

1. What are two most general PL categories?
2. What are the main reasons for the development of a new PL?
3. Which PL feature is concerned with the abstraction level supported in a program?
4. Which PL feature is concerned with the accessibility to the PL specification by the platform vendors?
5. Why should open source be factor in a PL's development plan?
6. Which of the PL features may have conflicting requirements? Briefly discuss.
7. What are the main reasons for programmers to study a PL properly?

Chapter 1 Exercise

The main objective of the exercises in this chapter is to review the basis of Java programming. This is needed to learn other Java programming concepts and techniques in the rest of the book.

1. Working with a Java IDE.

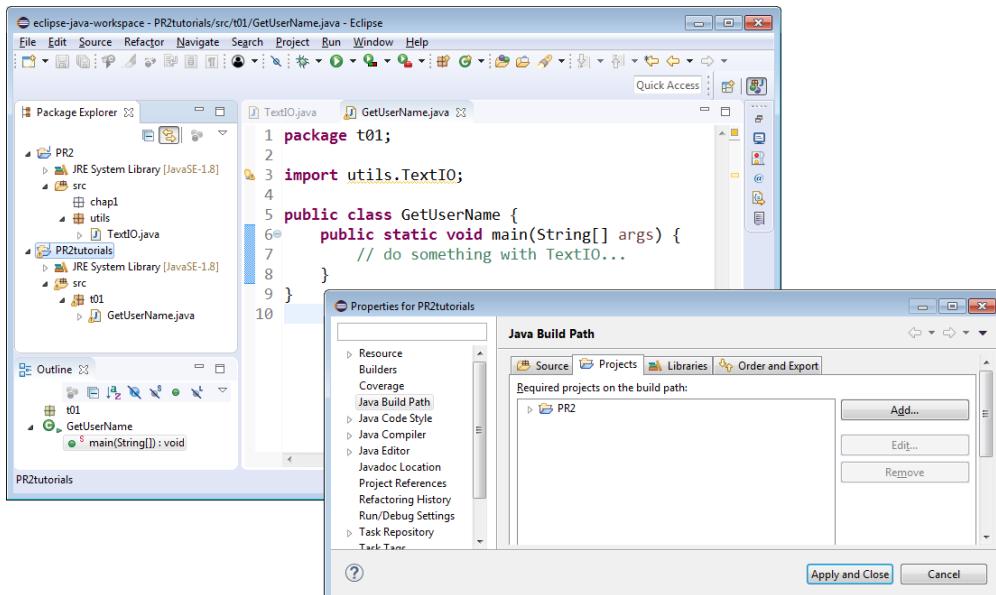


Figure 1.3: Structuring a Java project in Eclipse.

Starting from this chapter, you will use an **Integrated Development Environment (IDE)** to write, compile and run your Java programs. This is easier than having to manually use the commands `javac` and `java` on the terminal. In this book, we will use the Eclipse IDE¹². Other commonly used IDEs include Netbeans¹³ and IntelliJ IDEA¹⁴.

Most IDEs use the concept of a **project** to organise your Java programs. A project is basically a collection of related Java programs. Generally, you will need to *create two projects*: one for the source code that you will be given in the lectures and the other is for tutorials.

Open Eclipse from the Programs menu of your computer. Create two projects: (1) project `src` for holding the source code of the lectures and (2) project `tutorials` for the tutorials. For example, Figure 1.3 shows how to achieve this in Eclipse. Note how project `PR2tutorials` reference the `PR2` project in its “Java Build Path” properties. You will use project `tutorials` for doing the exercises. For example,

¹²<https://www.eclipse.org>

¹³<https://netbeans.org>

¹⁴<https://www.jetbrains.com/idea/>

create a package named `tute1` for week 1's exercises, `tute1` for week 2, and so on.

- 2. Working with source code package `utils`.** The source code that comes with this book, which hereforth will be referred to as the **attached source code**, use of a number of shared Java classes in a package named `utils`. These are utility classes that you would need to use for some of the exercises. For this chapter, for instance, you will need to use a class named `TextIO.java`.

Extract the package `utils` from the attached source code and follow the instructions of your IDE to create this package in your project. In a class that needs to use `TextIO`, add one of these lines (depending how you want to call the methods of this class):

```
import static utils.TextIO.*;
```

OR

```
import utils.TextIO;
```

- 3. Working with class `TextIO`.** The following exercises in [6] help practise class `TextIO`. You will do these exercises as part of the Java review (below).

- Chapter 2: 2.3-5 and 2.7.
- Chapter 3: 3.3-5.

4. Working with basic data types (1).

Listing 1.5: Program Interest

```
1  /**
2  * This class implements a simple program that
3  * will compute the amount of interest that is
4  * earned on 17,000 invested at an interest
5  * rate of 0.07 for one year. The interest and
6  * the value of the investment after one year are
7  * printed to standard output.
8 */
9 public class Interest {
10     public static void main(String[] args) {
11         /* Declare the variables. */
12         double principal;
13         double rate;
14         double interest;
15         /* Do the computations. */
16         principal = 17000;
17         rate = 0.07;
18         interest = principal * rate;
19         principal = principal + interest;
```

```

20  /* Output the results. */
21  System.out.print("The interest earned is ");
22  System.out.println(interest);
23  System.out.print("The value of the investment after one year is "
24      );
25  System.out.println(principal);
26 } // end of main()
} // end of class Interest

```

Answer the following questions about program Interest:

- (a). Change the statement at line 16 to the following statement. Compile and run the program again. Does it work? Why do you think that is?

```
principal = 17000.0;
```

- (b). *Change the statement at line 12 to the following statement. Compile the program again. Does it work? Why do you think that is? Can you fix it without reversing the change?

```
int principal;
```

5. Working with basic data types (2).

Do exercises 2.1-6 in [6].

6. Working control statements.

Do exercises 3.1-4 in [6].

7. Working with files.

Do exercises 2.7 and 3.5 in [6].

8. Working with arrays (1).

Listing 1.6: Program Array

```

1 /**
2  * This class implements a simple program that
3  * will initialise an array, performs some
4  * operations on it and prints the results on the
5  * standard output
6 */
7 public class Array {
8     public static void main(String[] args) {
9         /* Declare the variables. */
10        int nums[] = { 2, 0, 1, 3 };
11        /* Do some operations on array */
12        int sum = 0, i;
13        int n;
14        for (i = 0; i < 4; i++) {
15            n = nums[i];
16            System.out.printf("nums[%d] = %d\n", i, n);
17            sum = sum + n;
18        }

```

```

19     System.out.print("Sum: ");
20     System.out.println(sum);
21 } // end of main()
22 } // end of class Array

```

Answer the following questions about program Array:

- (a). What does the statement at line 10 do?
- (b). Java supports another (more intuitive) form of array initialisation. Change the statement at line 10 to this statement: int [] nums = { 2, 0, 1, 3 }; Compile and run the program. Does it work? What is the difference here?
- (c). What does System.out.printf at line 16 do?
- (d). Change the for loop at lines 14-18 so that it calculates two sums: the sum of the elements at the even indices and the sum of the elements at the odd indices. Then, change the print statements at lines 19-20 so that the program prints out both sums on the standard console.

9. Working with arrays (2). Do exercise 7.2 in [6].

10. Working with strings.

Listing 1.7: Program Strings

```

1 import java.util.Arrays;
2 /**
3  * This class implements a simple program that
4  * will initialises a string, performs some
5  * basic operations on it and prints the results on the
6  * standard output
7 */
8 public class Strings {
9     public static void main(String[] args) {
10         /* Declare the variables. */
11         String str = "to be or not to be";
12         /* Do some operations on string */
13         char chars[] = str.toCharArray();
14         // convert the array to string
15         String charsAsString = Arrays.toString(chars);
16         int len = str.length();
17         String w1 = str.substring(3,5);
18         String w2 = str.substring(16,18);
19         boolean equal = w1.equals(w2);
20         /* Print out the results */
21         System.out.println("string: " + str);
22         System.out.println("length: " + len);

```

```
23     System.out.println("word 1: " + w1);
24     System.out.println("word 2: " + w2);
25     System.out.println("word 1 is equal to word 2? " + equal);
26     System.out.println("characters: " + charsAsString);
27 } // end of main()
28
29 } // end of class Strings
```

Answer the following questions about program `Strings`:

- (a). What is the output of `str.length()`?
- (b). What are words `w1` and `w2` and what is the result of `w1.equals(w2)`?
- (c). What do the two arguments (3,5) and (16,18) of the two invocations of `str.substring()` mean?
- (d). Change the equality test at line 19 to the following statement. Compile and run the program again. What is the result? Why do you think it is?
`boolean equal = (w1 == w2);`
- (e). Given the usage of `toCharArray`, what is another way of determining the number of characters in the string `str`? Using comments, briefly write the instructions that you would write in the `main` method to make this happen.
- (f). What is the effect of the operator `+`?
- (g). What would the output at the last line tell you about strings in Java?

Chapter 2

The Java Programming Language

Objectives

- ✓ Understand programming language syntax and semantics and the relationship between them
- ✓ Explain the context-free grammar rule structure and a diagram notation for representing this structure
- ✓ Describe the Java language syntax rule
- ✓ Explain the syntax rules of the core Java language constructs, including identifier, method declaration, block, statement and expression

In this chapter, we will study the Java programming language specification with an aim to understand its features at a depth below the usage level. We will discuss language and semantics as two components of a language specification and focus on the Java syntax. We will learn the grammar structure of and how to interpret the Java syntax. We illustrate the syntax using some of the core language constructs (identifier, method declaration, block, statement and expression).

2.1 Motivation

We discussed in Chapter 1 the need for the programmer to study a PL's features to a level that would allow her to make informed decisions about which feature(s) are most suitable for solving a particular problem. To achieve this, however, requires knowing all the alternative features and the pros and cons of each. For example, let us consider questions such as “how many ways can an array be defined?” and “how many ways can a method be defined?”.

We can learn the different ways through reading code examples about arrays and methods. Although this is the approach that is most common among programmers when they learn to use a new language, it can not help find satisfactory answers to the questions above. The only approach that does is to study the language specification syntax about array and method declarations. This syntax, much like the natural language syntax, describes the valid grammar rules for each language construct. These rules define the set of all the valid sentences that can be written in the language concerning the construct.

In this chapter, we will briefly introduce this approach and demonstrate it with the Java language.

2.2 Language Syntax and Semantics

The specification of a PL, much like natural language (NL), consists in two parts: syntax and semantics. In this section, we briefly introduce these two components and the relationship between them. We will focus primarily on syntax and, in particular, discuss the syntax grammar.

2.2.1 Syntax

The Cambridge online dictionary has two definitions for syntax¹. First in the NL context, it is “... the grammatical arrangement of words in a sentence”. Second in the computing context, it is defined somewhat similarly as “... the structure of statements or elements in a computer language”. More generally, a language syntax [15, 28] consists in a set of rules that specify how sentences are constructed from a given alphabet (the valid characters set). The construction is typically defined incrementally: from characters to words then words to sentences. Let us illustrate this with an analogy in NL. After that, we will look at a PL example.

Example 2.1 Natural language text

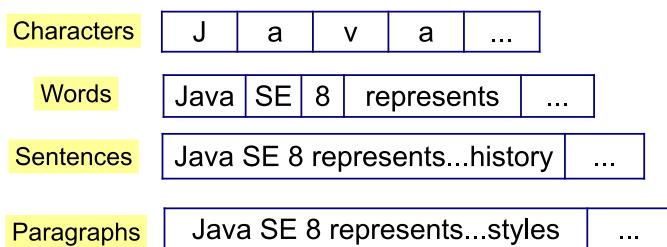


Figure 2.1: Syntax analysis of the example text.

Consider this excerpt from the preface of the Java language specification [11]: “Java SE 8 represents the single largest evolution of the Java language in its history. A relatively small number of features - lambda expressions, method references, and functional interfaces - combine to offer a programming model that fuses the object-oriented and functional styles”. This paragraph is broken down into sentences, words and characters as shown in Figure 2.1. □

¹<https://dictionary.cambridge.org/dictionary/english/syntax>

Let us next consider an example Java program and, for illustration purpose, uses the NL's approach above to mechanically break down its text into elements. We will study a precise approach for the Java language syntax later in this chapter.

Example 2.2 Java program text

Listing 2.1: Java program Gcd.java

```

1 public static void main(String[] args) {
2     int i = TextIO.getInt(),
3         j = TextIO.getInt();
4     while (i != j) {
5         if (i > j) i = i - j;
6         else j = j - i;
7     }
8     TextIO.putln(i);
9 }
```

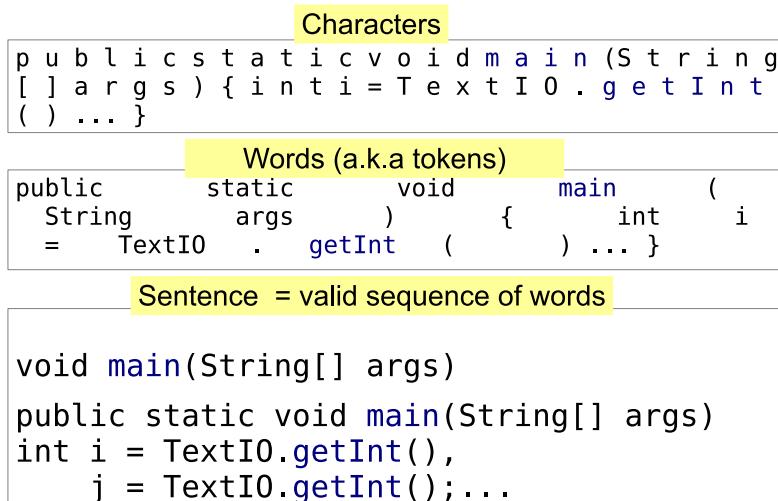


Figure 2.2: Syntax analysis of the Java program Gcd in Listing 2.1.

The Java program named Gcd presented in Listing 2.1 is broken down into sentences, words and characters as shown in Figure 2.2. □

2.2.2 Syntax Grammar

Technically, PL syntax is defined using grammar – a set of rules that govern how language elements are formed. It is important for a PL syntax to be precise so that programming statements (the ‘sentences’) written in the language are unambiguous to the computer and can therefore be executed. Although there are many PLs, most of them are defined using one of two types of grammar: regular and context-free grammars [28]. Further, these two grammars are defined in terms of four basic types of rules:

- *concatenation*: sequentially combining elements
- *selection* (*one of*, a.k.a *alternation* [28]): selection among a finite set of elements
- *repetition* (a.k.a *Kleene closure* [28]): repeating an element a number of times
- *recursion*: defining an element in terms simpler elements of the same type

Regular grammar consists of regular expressions that are defined in terms of the first three rules, while context-free grammar (CFG) is constructed using all four types of rules. We will focus on CFG in this book. Before illustrating these grammars, let us briefly introduce the grammar rule notation.

Grammar rule notation

A grammar rule defines a symbol or a token. A token is a string that has well-defined meaning. When a token's definition requires more than one rules, exactly one of these rules defines the token. The remaining rules define the components of the token. Each of these components is labelled by a symbol.

In general, a grammar rule is written using this form:

$LHS \rightarrow RHS$

- left-hand-side (LHS): a symbol or token, whose content needs to be defined
- right-hand-side (RHS): describes the content of the LHS, which is a sequence of one or more other symbols
- \rightarrow (arrow): means “has the form” or “goes to” [28]. In the Java syntax specification [11], this arrow is replaced by colon ‘:’

In addition to this basic form, the notation specifies the writing convention for the four rule types in the RHS as follows:

- concatenation: sequentially listing the elements, separated by spaces ‘ ’
- selection: (*one of*) followed by the sequential listing of elements
- repetition: $\{x\}$, i.e. repeat x zero or more times²
- recursion: simply including the LHS as part of an element of the RHS

To process the grammar rules of a token, the RHS of the token rule is expanded by repeatedly replacing each symbol with its corresponding RHS (if any). This process is stopped when there are no more RHS to be expanded.

CFG terminology

In CFG, in particular, each rule is called a *production* and the LHS symbol of every production is called a *nonterminal*. The nonterminal of the first production is called the

²Scott [28] suggests c^* , which uses the Kleene star ‘*’.

start symbol, because it is the starting point of and, thus, represents the entire grammar. The opposite of nonterminal is the type of symbol, called *terminal*. This includes the pre-defined tokens of the grammar, which are not defined by productions. Terminals can only appear on the RHS of productions.

Example 2.3 Regular and context-free grammars

To illustrate regular and context-free grammars, let us build two example languages, one with each type of grammar. The first example is the regular grammar shown in Listing 2.2, which is adopted from Chapter 2 of [28]. This grammar constructs a language of natural numbers. It is a regular grammar because no rules use recursion, that is the LHS of every rule does not appear in the RHS of any other rules.

Listing 2.2: Regular grammar for natural numbers

```

1 digit → (one of) 0 1 2 3 4 5 6 7 8 9
2 non_zero_digit → (one of) 1 2 3 4 5 6 7 8 9
3 natural_number → non_zero_digit {digit}

```

The second example is the CFG shown in Listing 2.3. In addition to the three basic rules of regular grammar, this grammar uses recursion in the third rule to define `binary_number` in terms of itself (appearing also in the RHS). It can be seen without difficulty that this grammar defines a language containing the following subset of binary numbers: $\{10, 11, 1010 \dots, 1111 \dots, 10111011 \dots, 11101110 \dots\}$.

Listing 2.3: CFG of a binary numbers subset

```

1 binary → (one of) 0 1
2 starts_one → 1 binary
3 binary_number → (one of) starts_one {binary_number}

```

□

Example 2.4 Java syntax grammar

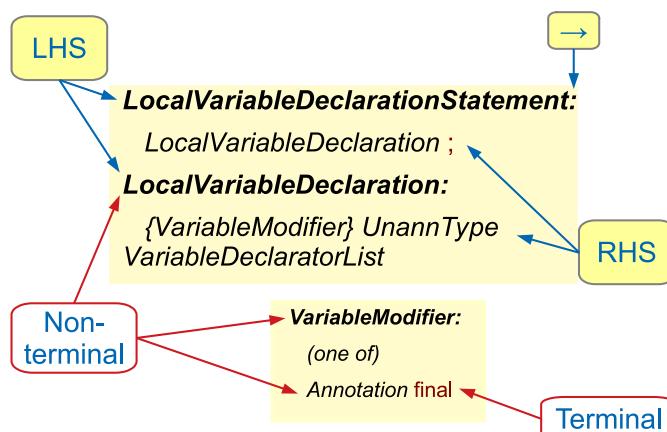


Figure 2.3: The context-free grammar rule for variable declaration in Java.

To complete the syntax grammar example in this section, we show in Figure 2.3 the Java syntax rule for the local variable token. We will discuss the Java syntax in more detail later in this chapter. As shown in the figure, the token name is `LocalVariableDeclarationStatement`, which is defined in terms of the symbol `LocalVariableDeclaration` followed by ‘;’. The grammar rule for this symbol is also displayed in the figure. There are other rules that define components of this rule. For brevity, only one of these (`VariableModifier`) is shown in the figure. □

2.2.3 Syntax Tree

When a program is processed for execution (typically by a component called *parser*), the program statements are matched with the productions in the language grammar. This process effectively instantiates the matching productions using the program’s terms. The result is a **syntax tree (SYT)** which is later processed for execution. In Chapter 3, we will describe parsing as part of the general compilation process. For our purpose here, it suffices to know that SYT helps ease the program analysis by visually presenting its structure for inspection. We will use SYT later in this chapter to visually present the syntax structures of the program examples.

SYT has the following characteristics:

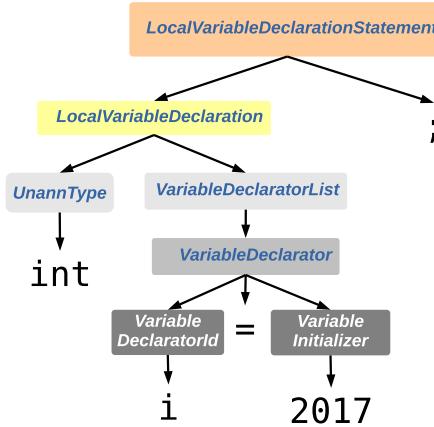
- **Node:** a terminal or nonterminal term that matches a statement element. Node label is the term name (in the JLS). Different nodes may have the same label if they match the same term
 - root node represents the first token of the program
 - leaf nodes represent keywords, separators or operators
- **Edge:** directed binary edge that connects two nodes representing two terms on two sides of a production rule. The edge direction is the same as the production rule arrow.

More specifically, given a production rule: $LHS \rightarrow RHS$ and a node n_1 that represents LHS. We form an edge (n_1, n_2) with every node n_2 that represents a terminal or nonterminal in the RHS.

- Edges of the same rule are arranged from left to right, in the same order of terms in the RHS
- The number of edges defined for one production rule equals the number of terms in the RHS that match the code statement

Example 2.5 Syntax Tree

Figure 2.4 shows the SYT of the following code statement:

**Figure 2.4:** An example syntax tree.

```
int i = 2017;
```

Token LocalVariableDeclarationStatement is the root node because it is mapped to the top-level statement of the program. \square

2.2.4 Nested box diagram – An Alternative Representation of Syntax Tree

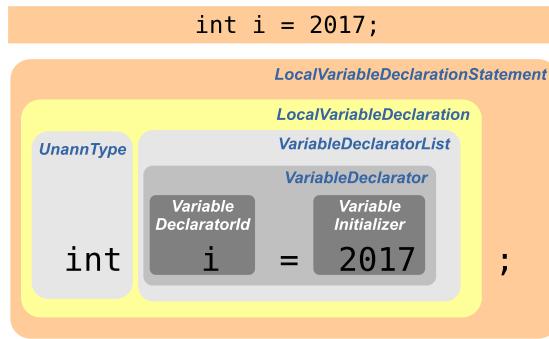
The syntax tree is generally useful for representing the detailed program structure. However, its structure is not natively mapped to the program text. For presentation purpose in this chapter, we introduce another visual presentation of program structure, which we call **nested block diagram (NBD)**. The nested structure of NBD fits naturally with the containment structure of the CFG rules. We can draw this diagram superimposed on the program text, making it easier to make a connection between the text and its grammar structure. We will use NBD in our illustrations of the program examples in this chapter.

We represent each production in terms of an NBD as follows:

- LHS becomes the outer block
- RHS's element(s) become the inner block(s)
- production (\rightarrow) becomes block containment
- element sequence in the RHS is mapped to this inner block positioning scheme:
left to right and top to bottom

Example 2.6 Nested block diagram

Figure 2.5 shows the NBD of the syntax tree in Example 2.5. The outer-most block is mapped to the root node of the syntax tree (LocalVariableDeclarationStatement). This block contains two inner blocks, one for LocalVariableDeclaration and the other is for ‘;’. The former is further nested by another NBD which represents the

**Figure 2.5:** An example NBD.

production for `LocalVariableDeclaration`. The nesting is stopped when we reach the leaf nodes that represent the program text elements. \square

2.2.5 Semantics

In general, the semantics component of a PL specifies the meaning of a program written in the language. In principle, this meaning defines the values that are assigned to the program symbols. Without these values, it is impossible to differentiate between different symbols in the same language and to produce the expected program output.

A NL sentence may be grammatically correct but does not make a valid sense (i.e. incorrect meaning). What determines correctness depends on the context of use. Similarly, syntactically-correct program may be semantically wrong. Let us illustrate these with an NL example and a Java program example.

Example 2.7 A semantically-wrong NL sentence

In the context of the JLS excerpt presented in Example 2.1, the following sentence has an incorrect meaning, despite the fact that its syntax is valid:

“Java SE 8 represents the *evolution* of the *English language*.”

The two semantic errors highlighted in the sentence are: (1) “evolution” (should be a “single largest evolution”) and “English language” (should be “Java language”). \square

Example 2.8 A semantically-wrong Java program

In the context of the program Gcd of Example 2.2, the following Java statement is incorrect, despite the fact that it does not violate any Java syntax rules. As such, this statement will be reported at compile time as have errors.

```
Void main(String[] args)
```

The problem here is the word `Void`, which is neither a Java’s keyword nor a user-defined data type. Since it has not yet been defined, the Java compiler has no way of assigning values to it and thus it will report as an error.

Let us consider another statement:

```
int i = TextIO.getInteger(), j = TextIO.getInt();
```

which has a semantic error in the initialisation of variable *i*. The error is the method name `getInteger`, which is not defined in the class `TextIO`. The correct method name is `getInt`, which is used in the initialisation of the variable *j*. \square

In programming, semantics is checked at compile time and/or at run time. The semantic rules that are checked at compile time form the *static semantics* of the language. The compiler is responsible for enforcing static semantics. The remaining semantic rules that are checked at run time form the *dynamic semantics*. Example rules that can only be checked at run time are division by zero and accessing an out-of-bound array index. These rules only occur with certain cases of the program variables.

2.3 The Java Language Syntax

In this section, we introduce the structure of the Java language syntax and illustrate it with a number of core constructs. The main aim is to explain how to read the JLS so that its syntax rules can be looked up and studied when needed. Much of the material presented here are taken from the Java language specification (JLS) [11]. To ease notation, we will use $\S X$ to refer to a section *X* in the JLS.

JLS defines the synax using two sets of grammars: lexical grammar and syntactic grammar. Lexical grammar defines how characters are combined to form valid tokens. Syntactic grammar, on the other hand, describes how tokens are combined to form valid programming statements. Both grammars are CFG.

2.3.1 Lexical grammar

Java's lexical grammar is defined in §3, which has the following characteristics:

- *start symbol* is Input (§3.5)
- *terminal symbols* are the Unicode character set
- *productions* describe how sequences of Unicode characters (§3.1) are translated into a sequence of input elements (§3.5)
- *tokens* are input elements with whitespace and comments removed. These include:
 - identifiers (§3.8): an unlimited-length sequence of Java letters and Java digits, thefirst of which must be a Java letter
 - keywords (§3.9): 50 reserved words
 - literals (§3.10): the source code representation of a value of a primitive type, string, or null type

- separators (§3.11): 12 tokens
- operators (§3.12): 38 tokens

Listings 2.4 and 2.5 list the grammar rules for separators, operators and keywords.

Listing 2.4: Java separators and operators

Separator:

```
(one of)
( ) { } [ ] ; , . . . @ ::
```

Operator:

```
(one of)
= > < ! ~ ? : ->== >= <= != && || ++ --+ - *
/ & | ^ % << >> >>>+= -= *= /= &= |= ^= %=

<<= >>= >>>=
```

Listing 2.5: Java keywords

Keyword:

```
(one of)
abstract continue for new switch
assert default if package synchronized
boolean do goto private this
break double implements protected throw
byte else import public throws
case enum instanceof return transient
catch extends int short try
char final interface static void
class finally long strictfp volatile
const float native super while
```

2.3.2 Syntactic grammar

The syntactic grammar is the main part of the JLS as it defines the set of all the possible valid programming statements. Indeed, its content covers seven top-level sections of the JLS (§4.6-10,14,15). We will study a number of core rules of this grammar later in this chapter. The syntactic grammar has the following characteristics:

- *start symbol* is CompilationUnit (§7.3)
- *terminal symbols* are tokens produced by lexical grammar

- *productions* define the nonterminals that represent a broad range of language constructs

2.3.3 Additional Java grammar notation

In addition to the general CFG notation explained in Section 2.2.2, JLS uses the following notation to widen the applicability of the syntax:

- Nonterminal: written in italic font
- a multi-line RHS presents alternative definitions for the LHS (each line is one alternative)
- [x]: zero or one occurrence of x
- but not: exclude the following expansions

2.4 Identifier

In this and the remaining sections of this chapter, we will study the syntax structures of a number of core Java constructs. You are probably already familiar with these constructs from previous programming courses. However, since our objective here is to dive deeper into the syntax of the language, you are invited to revisit these constructs by going through these materials.

Definition 2.1. Identifier

An identifier (§3.8) is an unlimited-length sequence of Java letters and Java digits, the first of which must be a Java letter.



The production for identifier is shown in Listing 2.6.

Listing 2.6: Identifier grammar

Identifier:

IdentifierChars but not (Keyword BooleanLiteral NullLiteral)

IdentifierChars :

JavaLetter {JavaLetterOrDigit}

JavaLetter :

(one of) ‘_’ ‘\$’ NonDigitUnicodeLetter

JavaLetterOrDigit :

(one of) JavaLetter Digit

Digit :

(one of) 0 1 2 3 4 5 6 7 8 9

Note that the productions for Javaletter and JavaLetterOrDigit have been rewritten from the informal definition in JLS [11] to conform to the CFG notation.

Example 2.9 Identifier

According to the identifier's definition, the following strings are valid identifiers: id, id1, _id1 and \$id1. However, these strings are invalid identifiers: 1id, i!d, @id and #id. □

2.5 Method Declaration

Definition 2.2

A method (§8.4) declares executable code that can be invoked, passing a fixed number of values as arguments.



Method is a class member, whose definition is captured in the production in Listing 2.7.

Listing 2.7: Method declaration grammar

```

MethodDeclaration:
    {MethodModifier} MethodHeader MethodBody

MMethodHeader:
    Result MethodDeclarator [Throws]
    TypeParameters {Annotation} Result MethodDeclarator [Throws]

MMethodDeclarator:
    Identifier ( [FormalParameterList] ) [Dims]

MethodModifier:
    (one of)
    Annotation public protected private
    abstract static final synchronized native strictfp

Result:
    UnannType
    void

Throws:
    throws ExceptionTypeList

ExceptionTypeList:
    ExceptionType { , ExceptionType}

MethodBody:
    Block
    ;

```

FormalParameterList:

```

    ReceiverParameter
    FormalParameters , LastFormalParameter
    LastFormalParameter

```

FormalParameters:

```

FormalParameter { , FormalParameter}
ReceiverParameter { , FormalParameter}
FormalParameter:
{VariableModifier} UnannType VariableDeclaratorId

```

Example 2.10 Method declaration

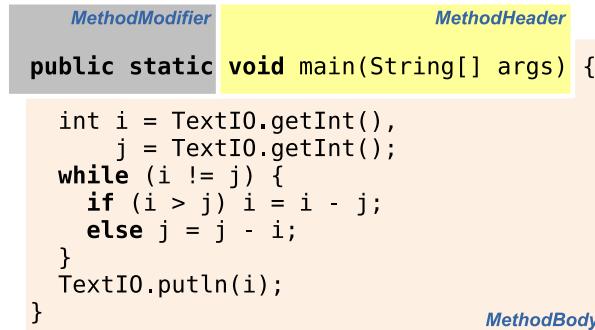


Figure 2.6: Method declaration example.

Figure 2.6 is the NBD for the declaration of a method `main`. The NBD shows structure of the top-level production rule for `MethodDeclaration`. Figure 2.7 shows the NBD for the method header part of the method declaration shown in Figure 2.6.

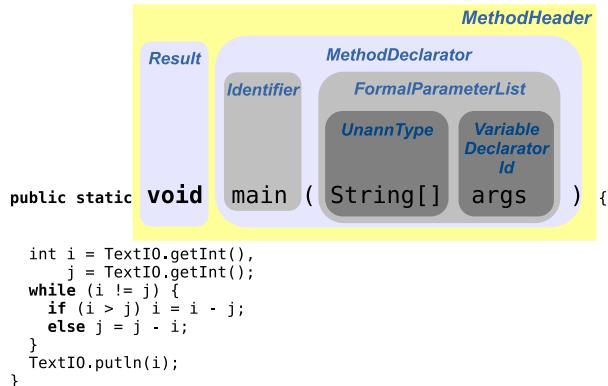


Figure 2.7: Method header example.

Figure 2.8 shows the NBD for another method declaration example that throws an exception. □

2.6 Blocks and Statements

Blocks and statements are defined in §14 of JLS. In this chapter, we will study the following constructs:

- Block (§14.2)
- Local variable declaration (§14.4)

```

MethodModifier           MethodHeader
public static void gcd(int i, int j)
throws IllegalArgumentException {  

if (i < 0 || j < 0) throw new  

    IllegalArgumentException(  

        "Gcd.gcd: (i,j)=(\"+i+\",\"+j+\")");  

while (i != j) {  

    if (i > j) i = i - j;  

    else j = j - i;  

}  

TextIO.putln(i);  

}

```

MethodBody**Figure 2.8:** Method declaration with exception.

- Statements:

- if (§14.9)
- while (§14.12)
- for (§14.14)
- switch (§14.11)

2.6.1 Block (§14.2)

Definition 2.3. Block

A block (§14.2) is a sequence of statements within braces.



The production rule for block is shown in Listing 2.8

Listing 2.8: Block grammar

<i>Block:</i>	{ [BlockStatements] }
<i>BlockStatements:</i>	BlockStatement {BlockStatement}
<i>BlockStatement:</i>	LocalVariableDeclarationStatement
	ClassDeclaration
	Statement

Local Variable Declaration (§14.4)

This production rule (partially shown in Listing 2.9) is used as part of block grammar, which states how local variables are defined within a block. The listing includes an important production rule for Java data type, named UnannType. This rule is reused to define many other production rules of the Java syntax.

Definition 2.4. Local variable

A local variable declaration statement declares one or more local variable names. 

Listing 2.9: Local variable declaration rules

```

LocalVariableDeclarationStatement:
    LocalVariableDeclaration ;

LocalVariableDeclaration:
    {VariableModifier} UnannType VariableDeclaratorList

VariableModifier:
    (one of)
        Annotation final

UnannType:
    UnannPrimitiveType
    UnannReferenceType

UnannPrimitiveType:
    NumericType
    boolean

UnannReferenceType:
    UnannClassOrInterfaceType
    UnannTypeVariable
    UnannArrayType

```

Listing 2.10: Local variable declaration list

```

VariableDeclaratorList:
    VariableDeclarator {, VariableDeclarator}

VariableDeclarator:
    VariableDeclaratorId [= VariableInitializer]

VariableDeclaratorId:
    Identifier [Dims]

VariableInitializer:
    Expression
    ArrayInitializer

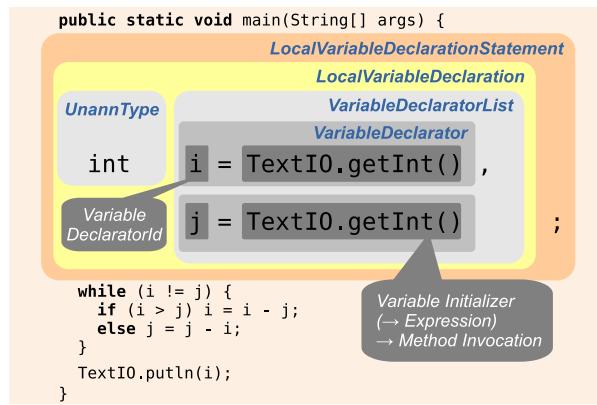
ArrayInitializer:
    { [VariableInitializerList] [ , ] }

VariableInitializerList:
    VariableInitializer { , VariableInitializer}

```

Example 2.11 Local variable declaration

Figure 2.9 shows the NBD for a local variable declaration statement of two variables *i* and *j* in the block forming the body of the same method *main* that we used in Figure 2.6 of Example 2.10. 

**Figure 2.9:** Local variable declaration example.

2.6.2 Statement

Java requires that the `else` clause of an `if` statement belongs to the innermost `if` statement. This is to support the case of an nested `if` statement. This leads to two rules for statement: `statement` and `statement-no-short-if`. These are defined in Listing 2.11.

Listing 2.11: Statement

Statement:

- StatementWithoutTrailingSubstatement
- LabeledStatement
- IfThenStatement
- IfThenElseStatement
- WhileStatement
- ForStatement

StatementNoShortIf:

- StatementWithoutTrailingSubstatement
- LabeledStatementNoShortIf
- IfThenElseStatementNoShortIf
- WhileStatementNoShortIf
- ForStatementNoShortIf

StatementWithoutTrailingSubstatement:

- Block
- EmptyStatement
- ExpressionStatement
- AssertStatement
- SwitchStatement
- DoStatement
- BreakStatement
- ContinueStatement
- ReturnStatement

SynchronizedStatement

ThrowStatement

TryStatement

LabeledStatement:

Identifier : Statement

LabeledStatementNoShortIf:

Identifier : StatementNoShortIf

2.6.3 if Statement

Definition 2.5

The *if* statement (§14.9) allows conditional execution of a statement or a conditional choice of two statements, executing one or the other but not both.



Listing 2.12: If statement

IfThenStatement:

if (Expression) Statement

IfThenElseStatement:

if (Expression) StatementNoShortIf else Statement

IfThenElseStatementNoShortIf:

if (Expression) StatementNoShortIf else StatementNoShortIf

Example 2.12 If statement

```
public static void main(String[] args) {
    int i = TextIO.getInt(), j = TextIO.getInt();
    while (i != j) {
        RelationalExpression (§15.20) IfThenElseStatement
        if ( i > j ) Expression ExpressionStatement
            i = i - j;
        else ExpressionStatement
            j = j - i; Assignment
                           Expression (§15.26)
    }
    TextIO.putln(i);
}
```

Figure 2.10: If statement example.

Figure 2.10 shows the NBD for an *if* statement in the body of the same method *main* that we used in Figure 2.6 of Example 2.10. This statement has an *else* clause. □

2.6.4 while Statement

Definition 2.6

The while statement (§14.12) executes a boolean Expression and a Statement repeatedly until the value of the Expression is false.



The production rule for the while statement is shown in Listing 2.13.

Listing 2.13: The while statement

```
WhileStatement:
    while ( Expression ) Statement
```

```
WhileStatementNoShortIf:
    while ( Expression ) StatementNoShortIf
```

Example 2.13 The while statement

Example

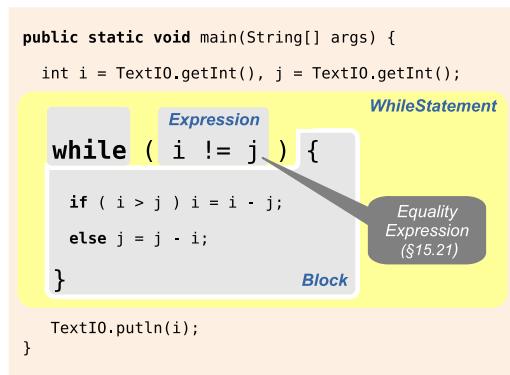


Figure 2.11: The while statement example.

Figure 2.11 shows the NBD for the while statement in the body of the same method `main` that we used in Figure 2.6 of Example 2.10. □

2.6.5 for Statement

The for statement (§14.14) has two forms: basic statement and enhanced statement.

2.6.5.1 Basic for Statement

Listing 2.14: Basic for statement

```
ForStatement:
    BasicForStatement
    EnhancedForStatement
```

```

ForStatementNoShortIf:
    BasicForStatementNoShortIf
    EnhancedForStatementNoShortIf

BasicForStatement:
    for ( [ForInit] ; [Expression] ; [ForUpdate] ) Statement

BasicForStatementNoShortIf:
    for ( [ForInit] ; [Expression] ; [ForUpdate] ) StatementNoShortIf

ForInit:
    StatementExpressionList
    LocalVariableDeclaration

ForUpdate:
    StatementExpressionList

StatementExpressionList:
    StatementExpression { , StatementExpression}

```

Example 2.14 Basic for statement

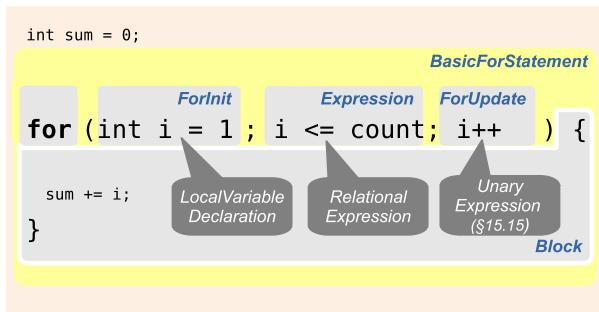


Figure 2.12: The for statement example.

Figure 2.12 shows the NBD for the basic for statement that computes the running total of a sequence of numbers. □

2.6.5.2 Enhanced for Statement

Definition 2.7. Enhanced for statement

The enhanced for statement is the foreach statement, which is convenient for iterating over a collection of elements.



Listing 2.15: Enhanced for statement

EnhancedForStatement :

```
for ( {VariableModifier} UnannType VariableDeclaratorId
      : Expression )
      Statement
```

EnhancedForStatementNoShortIf :

```
for ( {VariableModifier} UnannType VariableDeclaratorId
      : Expression )
      StatementNoShortIf
```

Example 2.15 Enhanced for statement

Example

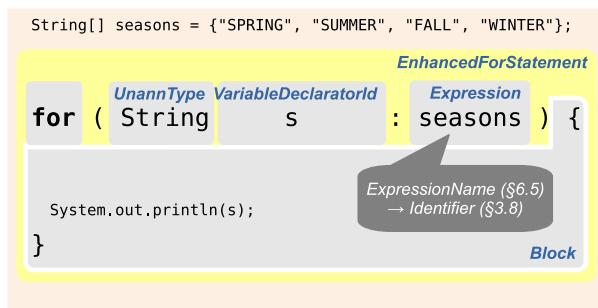


Figure 2.13: Enhanced for statement example.

Figure 2.13 shows the NBD for an enhanced for statement. This allows the program to iterate over an array of strings, representing season names, and print them to the standard console. □

2.6.6 switch Statement

Definition 2.8. The switch statement

The switch statement (§14.11) transfers control to one of several statements depending on the value of an expression.



The production rule for this statement is shown in Listing 2.16.

Listing 2.16: The switch statement

```
SwitchStatement :
  switch ( Expression ) SwitchBlock

SwitchBlock :
  { {SwitchBlockStatementGroup} {SwitchLabel} }

SwitchBlockStatementGroup :
  SwitchLabels BlockStatements
```

```

SwitchLabels:
  SwitchLabel {SwitchLabel}

SwitchLabel:
  case ConstantExpression :
  case EnumConstantName :
  default :

EnumConstantName:
  Identifier

```

Example 2.16 The switch statement Figure 2.14 shows the NBD for a switch statement

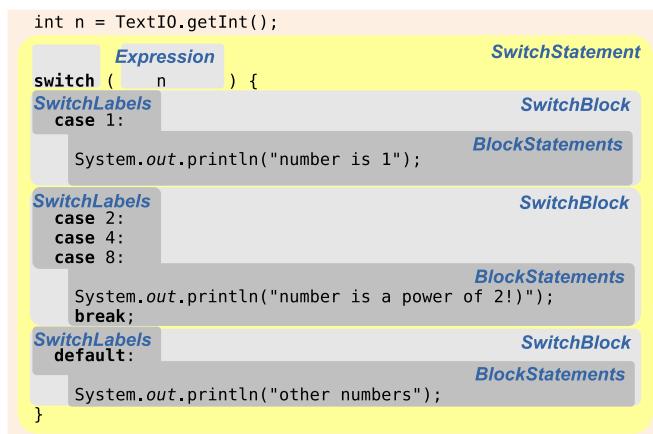


Figure 2.14: The switch statement example.

that prints to the console a message characterising an integer n .



Question What is/are missing from the diagram?

□

2.7 Expression

Definition 2.9

The result of evaluating expression denotes one of three things:

- *a variable (§4.12)*
- *a value (§4.2, §4.3)*
- *nothing (the expression is void): a method invocation that invokes a method that does not return a value (i.e. declared void)*



Expression has many different forms. In this section, we will focus on expression name and some forms of primary expression. The production rule is shown in Listing 2.17.

Listing 2.17: Expression name

ExpressionName :
 Identifier
 AmbiguousName . Identifier

AmbiguousName :
 Identifier
 AmbiguousName . Identifier

2.7.1 Primary Expression

Primary expression includes the following forms:

- **literals**
- object creations
- field accesses
- **method invocations**
- method references
- **array accesses**
- parenthesized expression

The production rule for primary expression is shown in Listing 2.18.

Listing 2.18: Primary expression

Primary :
 PrimaryNoNewArray
 ArrayCreationExpression

PrimaryNoNewArray :
 Literal
 ClassLiteral
 this
 TypeName . this
 (Expression)
 ClassInstanceCreationExpression
 FieldAccess
 ArrayAccess
 MethodInvocation
 MethodReference

2.7.2 Literal

Definition 2.10. Literal

A literal is the source code representation of a value of a primitive type (§4.2), the String type (§4.3.3), or the null type (§4.1).



The production rule for literal is shown in Listing 2.19.

Listing 2.19: Literal

Literal :

```
IntegerLiteral
FloatingPointLiteral
BooleanLiteral
CharacterLiteral
StringLiteral
NullLiteral
```

Example 2.17 Literal

Listing 2.20: Literal

```
1      101      0b0100
0.5   1.5f     2.5d
true  false
'a'   'A'     'g'     'Z'     '\u0041'   '\u0000'
"hello Java"  "principles"
null
```

Listing 2.20 shows several examples of literal. □

2.7.3 Method Invocation

Definition 2.11. Method invocation

A method invocation expression is used to invoke a class or instance method.



The production rule for method invocation is shown in Listing 2.21.

Listing 2.21: Method invocation

MethodInvocation :

```
MethodName ( [ArgumentList] )
TypeName . [TypeArguments] Identifier ( [ArgumentList] )
ExpressionName . [TypeArguments] Identifier ( [ArgumentList] )
Primary . [TypeArguments] Identifier ( [ArgumentList] )
super . [TypeArguments] Identifier ( [ArgumentList] )
TypeName . super . [TypeArguments] Identifier ( [ArgumentList] )
```

```
ArgumentList:
Expression { , Expression}
```

Example 2.18 Method invocation

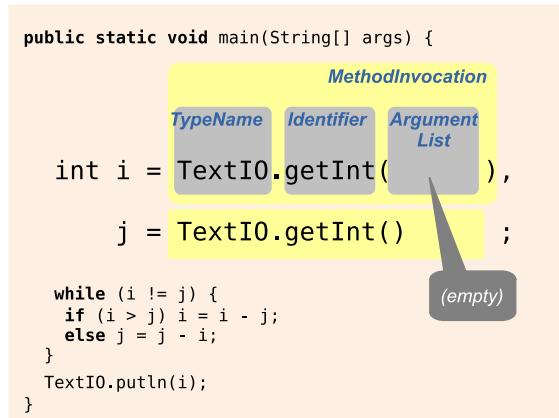


Figure 2.15: Method invocation without arguments.

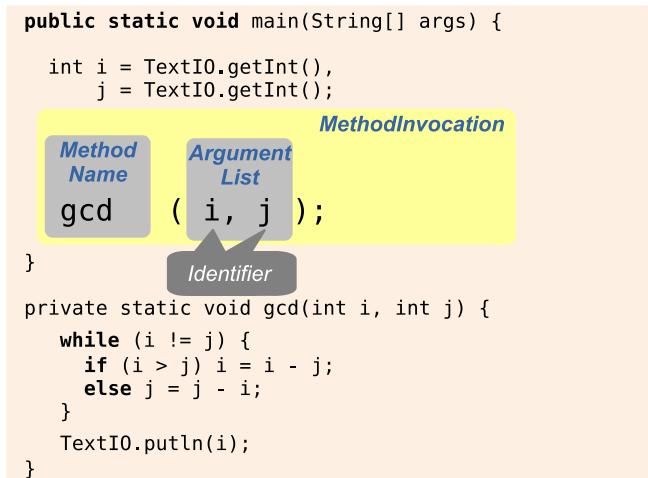


Figure 2.16: Method invocation with arguments.

Figures 2.15 and 2.16 respectively show an example of method invocation with and without arguments. □

2.7.4 Array Access

Definition 2.12. Array access

An array access expression refers to a variable that is a component of an array.

This component is selected by the value of the index expression.

It contains two subexpressions:

- array reference expression (before the left bracket): may be a name or any

primary expression that is not an array creation expression (§15.10)

- *index expression (within the brackets)*



The production rule for array access is shown in Listing 2.22.

Listing 2.22: Array access

ArrayAccess :

```
ExpressionName [ Expression ]
PrimaryNoNewArray [ Expression ]
```

Example 2.19 Array access

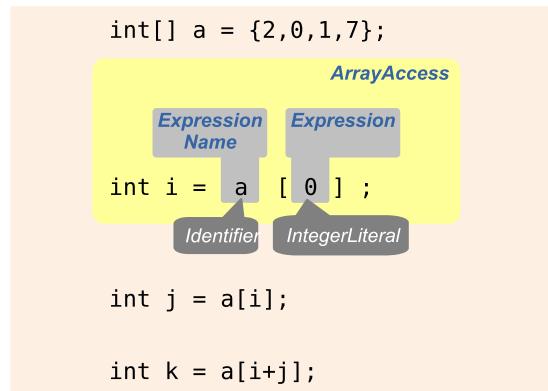


Figure 2.17: Array access example.

Figure 2.17 shows an example of array access for variable `i`.



Question Draw the NBDs for the array access parts of the variables `j` and `k`.





Chapter 2 Exercise

The objective of these exercises is to practise analysing the Java programming language syntax.

1. Use the syntax tree to explain the grammar of the following Java code. Be as specific as you can about the name of the nonterminal (LHS) that matches each code element.

```

1 private static float min(float a, float b) {
2     return a <= b ? a : b;
3 }
```

2. Use the grammar syntax tree to explain the grammar of the following Java code. Be as specific as you can about the name of the nonterminal (LHS) that matches each code element.

```

1 static float min(int[] a) {
2     // TODO: implements this
3 }
```

3. Use the grammar syntax tree to explain the grammar of the following Java code. Be as specific as you can about the name of the nonterminal (LHS) that matches each code element.

```

1 static int nextCount(int i) {
2     int j;
3     { j = i + 1;
4     }
5     return j;
6 }
```

4. Use the grammar syntax tree to explain the grammar of the following Java code. Be as specific as you can about the name of the nonterminal (LHS) that matches each code element.

```

1 private static float min(float a, float b) {
2     float min;
3     min = a <= b ? a : b;
4     return min;
5 }
```

5. Use the grammar syntax tree to explain the grammar of the following Java code. Be as specific as you can about the name of the nonterminal (LHS) that matches each code element.

```

1 private static float min(float a, float b) {
2     float min;
3     if (a <= b) {
4         min = a;
5     } else {
6         min = b;
7     }
8
9     return min;
10 }
```

6. Use the grammar syntax tree to explain the grammar of the following Java code.

Be as specific as you can about the name of the nonterminal (LHS) that matches each code element.

```

1 static int search(int[] a, int x) {
2     for (int i = 0; i < a.length; i++) {
3         if (x == a[i]) return i;
4     }
5     return -1;
6 }
```

7. Use the grammar syntax tree to explain the grammar of the following Java code.

Be as specific as you can about the name of the nonterminal (LHS) that matches each code element.

```

1 public static void main(String[] args) {
2     int[] a = {1, 1, 2, 3, 5, 8, 13};
3     int x = TextIO.getInt();
4     int isAt = search(a, x);
5     System.out.printf("%d is in %s at %d", x, a, isAt);
6 }
```

8. Are the following productions correct? Explain why or why not. If yes, what do you think is a benefit of this definition? If no, how do you fix them?

MethodDeclaration:

 MethodModifiers MethodHeader MethodBody

MethodModifiers:

 {MethodModifier}

9. Use the ***BinaryIntegerLiteral*** production of ***IntegerLiteral*** (§3.10.1) to determine if each of the following literals are correct. Briefly state why or why not.

1 0B001100010

2 0b001_110_001L

3 1b0_0_1L

4 OB012L

5 1B0L

6 OBO 0 1L

Chapter 3

Introduction to Compiler and Virtual Machine

Objectives

- ✓ Explain what a compiler is and its basic operations.
- ✓ Describe a virtual machine in general and the Java virtual machine, in particular.
- ✓ Explain the process involved in executing a Java program.

In this chapter, we will take a brief look at how computer processes programs that are written to run on it. Conceptually, the computer would automate the task of analysing the program text, which we manually performed on the Java programs in the previous chapter. A key component that is responsible for performing this task is the **compiler**. We will attempt to understand how the compiler works and, in particular, look at a modern type of compiler called **virtual machine** – the kind of compiler that is used for Java.

Compiler is a complex topic, a whole book has been written on it. It is typically studied in depth in a programming language design course. In the space of this one chapter, our purpose therefore is simply to understand what a compiler is and a high-level description of the compiler operations. These operations employ some of the fundamental algorithms that are often used in designing complex real-world programs.

3.1 Compiler

3.1.1 What Is Compiler?

Compiler is basically a program translation program! Why, because it needs to understand languages and, in principle, translate texts written in one language to another. A familiar analogy is Google translator, which is a translator for natural languages. The only difference with compiler is that it processes PLs, which are more structured.

Figure 3.1 gives an abstract view of the compiler. The **source program** is written in a source language (L_S), which is usually a high-level language (such as C, Java, etc.). The **target program** is written in a target language (L_T), which is a machine-dependent language (such as assembly) and is, therefore, executable in the computer hardware. This is indicated in the diagram by a dashed-line oval. During execution (which is also

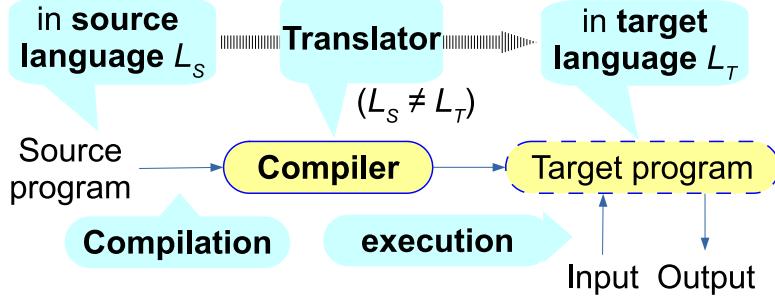


Figure 3.1: Overview of compiler (Adapted from [28]).

known as *run time*), the target program typically accepts certain inputs and generates some outputs.

While the abstract compiler view in Figure 3.1 is useful for conceptual understanding, it does not reflect the actual compiler implementation strategies. This is because, in practice, it is not often the case that there is a clear separation between the compilation (compile-time) and execution (run-time) of a program. While this is true for many traditional languages (e.g. C/C++ and Fortran), the separation is not absolute for more current languages (e.g. C#, Java, Perl, Python and Ruby). In scripting languages, such as Perl, Python and Ruby, some if not most of the compilation tasks are delayed until the program is executed. A more practical view, therefore, is to consider compilation as being performed at different levels.

3.1.2 Compilation Levels

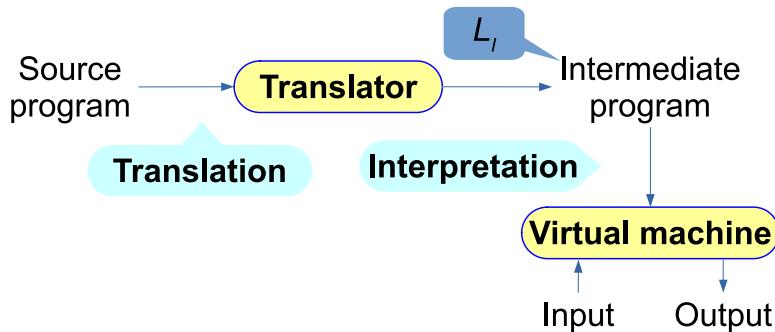


Figure 3.2: Compilation levels (Adapted from [28]).

Scott [28] suggests that most PLs employ a mixture of the two compilation levels shown in Figure 3.2. Some PLs place more emphasis on one level than on the other. Others, such as Java, focus on both levels.

Level 1 – translation: the source program is translated (by a *translator* component) into an intermediate program. This program is written in a machine-independent language (L_I), which can be run on different machines. The intermediate program is not

yet executable. It must be processed by another component called the *virtual machine* before it can be executed.

Level 2: interpretation: a **virtual machine (VM)** takes the intermediate program and the user input (if any), translates the intermediate program into a target program (in machine language) and executes it. Note that the target program is not displayed in Figure 3.2 because it is created as part of the VM's run-time environment and, as such, not written out physically to a file (like in traditional compilation).



Question What is a key benefit of the 2-level compilation?

Example 3.1 Compilation levels in Java VM

Listing 3.1: Program: HelloWorld

```

1 class HelloWorld {
2     public static void main(String[] args) {
3         System.out.println("Hello World!");
4     }
5 }
```

Listing 3.1 shows a simple hello world Java program. This program goes through two compilation levels as follows:

1. Translation:

- input: chap2/Hello.java
- translator: javac chap2/Hello.java
- intermediate program output: chap2/Hello.class

2. Interpretation:

- input: chap2.Hello (the chap2.Hello.class above)
- interpreter: java chap2.Hello
- output: Hello World!

We will study the two compilation levels of Java later in this chapter. □

3.1.3 Compiler Operations

Figure 3.3 gives an abstract view of the compiler operations. An operation, a.k.a a *phase* [28], transforms the program to some extent, making it ready to be processed by the next operation in the sequence. In the figure, the input and output of the operations are shown with the arrow lines. The symbol table is a shared repository for information about identifiers that are used throughout the compilation process.

The operations can be organised into two groups: *front end* and *back end*. The **front end** includes the first 3 operations, namely lexical, syntax and semantic analysis.

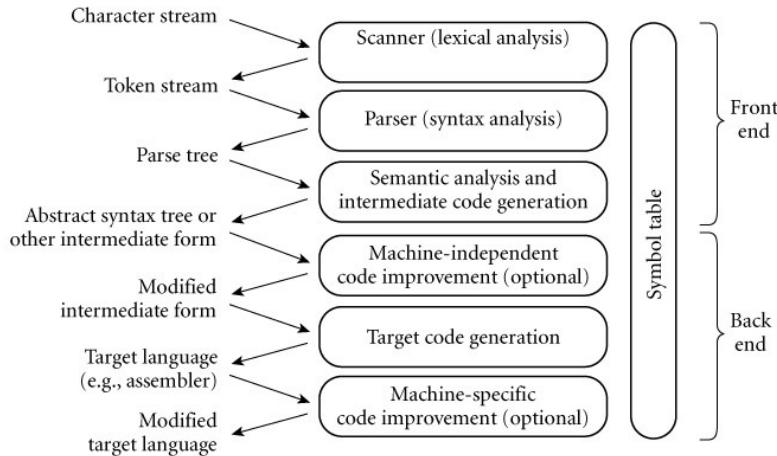


Figure 3.3: Compiler operations (Source: [28]).

The **back end** includes target code generation and code improvement.

 **Question** Do you see any familiar components (that you have used before in programming) being mentioned in the diagram?

Example 3.2 Program Gcd

In the remainder of this section, we will use the program Gcd in Listing 3.2 to illustrate the compiler operations.

Listing 3.2: Program Gcd (in C language)

```

1 int main() {
2     int i = getInt(), j = getInt();
3     while (i != j) {
4         if (i > j) i = i - j;
5         else j = j - i;
6     }
7     putInt(i);
8 }
```

□

3.1.4 Lexical and Syntactic Analysis

These two operations aim to validate the program structure, without paying attention to its meaning. Lexical analysis is performed by the **scanner** component, which accepts input in the form of a sequence (a.k.a stream) of characters. It produces tokens that are then used as input for syntactic analysis. Syntactic analysis, which is performed by the **parser** component, checks the program structure for any violation of the grammar rules. It generates as output a *syntax tree* (called *parse tree* or *concrete syntax tree* in [28]).

This is to be distinguished from another form of syntax tree, called abstract syntax tree, which is produced by the semantic analysis operation.

Any violations of the lexical and syntax grammar rules that are detected by the scanner and parser will result in compilation being stopped and error messages being displayed. A typical lexical error is invalid token; e.g. an identifier includes an invalid character. A typical syntactic error is invalid token sequence; e.g. no line separator character ‘;’ to end a statement.

Example 3.3 Lexical analysis

Listing 3.3: Lexical analysis of Gcd (Source: [28])

```

int      main    (          )      {      int i      =
getint   (          )      ,      j      =      getint
(
)
;      while   (          i      !=      j      )
{      if     (          i      >      j      )      i
=      i      -      j      ;      else   j      =
      j      -      i      ;      }      putint (
      i
)
;
}

```

Listing 3.3 shows the result of lexical analysis for program Gcd. From the character stream of the program text, namely (‘i’, ‘n’, ‘t’, ‘ ‘, ‘m’, ‘a’, ‘i’, ‘n’, …), the scanner produces a list of tokens. These include keywords (e.g. `int`, `if`, `while`, *etc.*), separators (e.g. ‘(’, ‘)’, *etc.*), identifiers (e.g. `main` `getint`, `i` and `j`) and operators (e.g. ‘>’, ‘=’, ‘!=’, *etc.*). \square

Example 3.4 Syntactic analysis: Gcd

Figure 3.4 shows the partial syntactic analysis for program Gcd. It displays two syntax trees for the statement `while (i != j)` (tree A) and the statement `putint (i);` (tree B). The arrow lines in both figures show mappings between the identifiers in both statements to the corresponding leaf nodes on the trees. \square

3.1.5 Semantic Analysis

This operation aims to determine the meaning of source program and generate the intermediate program. This program is transformed from the source program and commonly takes the form of an **abstract syntax tree (AST)**. The main difference between AST and the syntax tree produced by the parser is that all of the grammar-specific internal nodes are removed and that the leaf nodes are annotated with type information.

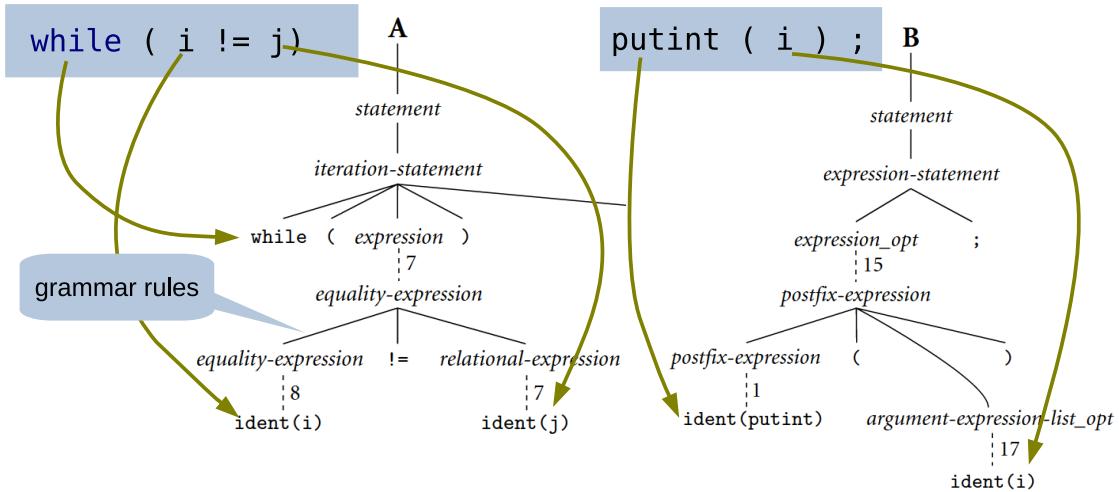


Figure 3.4: Partial syntactic analysis for the `while` and `putint` statements in Gcd (Adapted from [28]).

Semantic analyser records the data types of identifiers and expressions and ensure that they are used correctly throughout the program. To achieve this in a way that would assist the later operations to perform their tasks, the analyser constructs a shared data structure called **symbol table**. Each entry in the table maps an identifier to such information as data type, internal structure, and access scope. Example semantic rules that are checked for the C language using this table include [28]:

- identifier is declared before use
- correct use of identifier (e.g. using function name instead of a variable name in a function invocation)
- function call with correct parameter list
- a function that has a non-void return type actually returns a value

As discussed in Chapter 2, semantic rules can be checked either at compile time or at run time. Each alternative has its pros and cons. More compile-time check means less run-time check and, thus, better execution performance. In contrast, more run-time check means more execution flexibility but at the expense of a slower performance. In general, AST represents the intermediate program that is passed from the front end to the back end. In some specific compilers, other intermediate forms would be generated from the AST for the back end.

Example 3.5 Semantic analysis: Gcd.

Figure 3.5 shows the semantic analysis of program Gcd. The AST is displayed at the top and the symbol table is presented at the bottom left corner. The symbol table lists the symbols that appear in the AST together with their type definitions. For example, symbols #1 and #2 are data types (`void` and `int`, respectively), which are used to define the type definition of other symbols. Symbol #3 (`getint`) has a function type with one

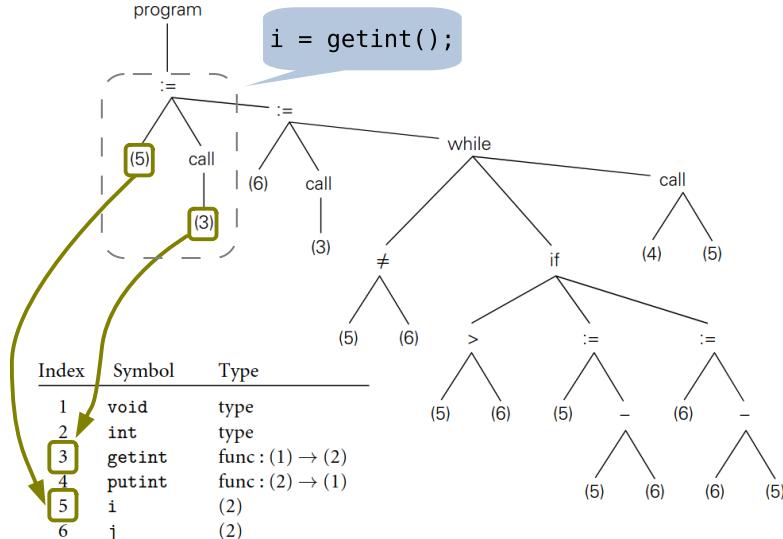


Figure 3.5: Semantic analysis of program Gcd (Adapted from [28]).

parameter (2) that has the type `int`. The AST differs from the syntax tree produced by the parser in Figure 3.4 in that the grammar-specific internal nodes are removed and that the leaf nodes are annotated with references to the corresponding identifier entries in the symbol table. \square

3.1.6 Target Code Generation

Once the program has been validated to be correct, it is ready to be translated into the target language. This translation is referred to as **target code generation** and the component that performs this is called **code generator**. The target language can be machine or assembly language. The latter is more readable by human (although much less so compared to such high-level languages as C and Java) and is less susceptible to machine platform update.

To generate the target code, the code generator basically uses the symbol table to rewrite the AST such that variables are replaced by memory locations and that suitable load and/or store operations are added for these variables. The rewrite also replace internal nodes of the AST by the corresponding arithmetic, test and branching operations of the target language. In assembly language, for example, memory locations are denoted by register names (e.g. `esp`, `ebp`, `eax`, `ebx` and `edi`). Examples of assembly language operations are `pushl` (push arguments of a function call on to a stack), `call` (call a function), `movl` (load/store variable), `cmpl` (compare variable) and `je` (jump/branch if comparing results in equal).

Example 3.6 Assembly code for Gcd

Figure 3.5 shows a partial assembly code that is generated for program Gcd. Note

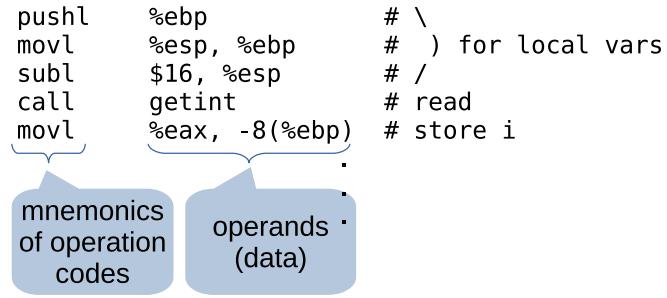


Figure 3.6: Partial assembly code for program Gcd (Adapted from [28]).

that `-8(%ebp)` refers to the memory location that appears 8 bytes before the location whose address is stored in `ebp`. \square

3.1.7 Code Improvement

In principle, the aim of **code improvement** (a.k.a *optimisation*) is to transform a program into an equivalent program (same behaviour) but can be executed more efficiently. Code improvement can be machine independent or machine dependent. The former performs the improvement on the intermediate program, while the latter does so on the target program. For example, a machine dependent code improvement that is performed on the assembly code of program Gcd in Example 3.6 would determine that the two variables `i` and `j` are only modified locally within the loop body and thus can be assigned directly to the registers. This helps eliminate the load and store instructions for these variables and, thus, makes the code significantly shorter and also executed faster.

3.2 Virtual Machine

As discussed in Section 3.1.2, **virtual machine (VM)** is the second component in the 2-level compilation model that interprets (and executes) the intermediate program. This component is the main part of the compilers of popular state-of-the-art scripting languages (e.g. PHP, Python, *etc.*) and also plays an important role in the compilers of popular nonscripting languages (e.g. Java and C#). In the remainder of this chapter, we will give an overview of VM and, in particular, the Java VM and how they work.

Conceptually, a VM is a software emulation of computer hardware, enough to allow programs to be executed on it. As such, the VM's API provides an instruction set architecture and necessary shared services for user programs. Among the key services are I/O, scheduling and memory management.

There are two types of VM: system VM and process VM [28]. **System VM** emulates the actual computer hardware and functions as a platform for executing alternative OSes.

It is often used to install ‘guest’ OSes on an existing (host) OS. An example system VM is the Oracle’s open-source Virtual Box¹.

Process VM emulates only a subset of the hardware functionalities, needed to execute programs at single-user processes. While system VM is tied to OS development, process VM is linked to PL development. Both state-of-the-art versions of Java and C# come with process VMs that allow programs to be easily portable to different platforms.

3.3 Java Virtual Machine

Java virtual machine (JVM) is a process VM that interprets the intermediate Java program and executes it in the computer. The intermediate programming language is called **byte code**, which is a platform-independent format suitable for transporting Java programs to different platforms. Figure 3.7 shows the relationships between JVM, the host platform and the programs that are run on it. JVM insulates programs from the details of the underlying host platforms so that the same program can be ported easily from one platform to another. There are different JVMs for different platforms, but only one program version is needed. JVM provides an API that enables programs to access and use the platform services.

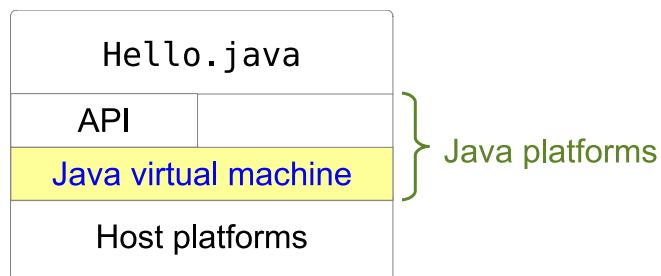


Figure 3.7: Java VM and platform.

As explained in Section 3.1.2 the Java platform performs a 2-level compilation for each Java program: compile (through the command `javac`) and interpretation (through the command `java`). Figure 3.8 illustrates these two steps with the simple Hello World program. Note that the Java’s byte code has been standardised so that the source program does not even need to be written in the Java language. The only requirement for the program to be executable on the JVM is that it strictly conforms to the byte code file format.

The initial versions of JVM were purely interpreted and were thus quite slow in performance. Starting from Java 2 (first released in 1998), JVM implements a **just-**

¹<https://www.virtualbox.org>

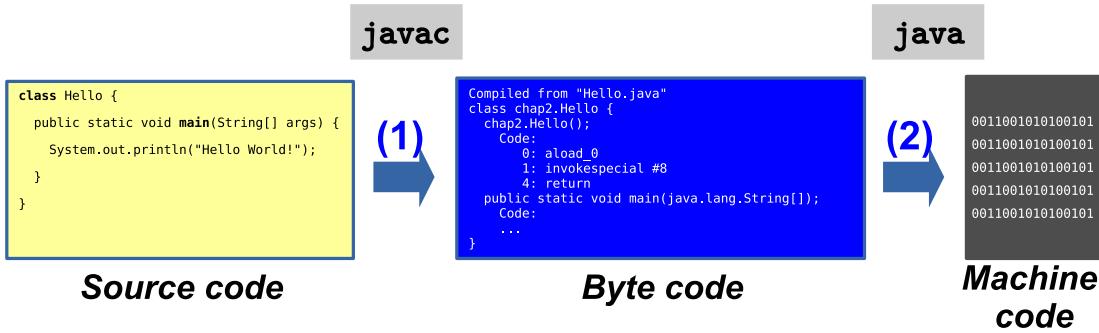


Figure 3.8: Java program development: (1) compilation, (2) execution.

in-time (JIT) compilation mechanism, by which the program byte code is interpreted immediately before each execution. How can that be achieved?

JIT employs a number of techniques to its front and back end operations. First, the front end operations are simplified because:

- *lexical analysis* is not needed because byte code is binary and not textual.
- *syntactic analysis* is simplified due to the descriptive nature of class file. Important structural properties (e.g. data types and parameter list) have already been added to the byte code by the source-to-byte-code translator.
- *code improvement* is simplified because certain improvements may already have been performed by the source-to-byte-code translator.

Second, performances of the back end operations are enhanced by using the following techniques:

- *lazy linker*: referenced libraries are only loaded and linked into the program when needed
- *dynamic (hot spot) compilation*: dynamically determines frequently used code paths and compile them directly into machine code. This is performed in parallel to executing the program and the compiled code is ready for execution when needed.
- *machine code caching*: the machine code of a class file may be cached for subsequent runs of the program.

3.3.1 Storage Management

The run-time data storage that are allocated by the JVM include two types [28]:

- **Global storage** is shared by all threads of execution and created and destroyed with the JVM. This includes method area, constant pool and heap
- **Per-thread storage** is created and destroyed with each thread. This includes base registers, method call stack and (optional) native method call stack.

A Java program potentially consists of one or more threads of execution. Each thread is an object of the class `Thread`. We will give a brief overview of the key storage areas below.

Method call stack and base registers

Method call stack, or **stack** for short, is a memory area holding information about method calls that are made by a thread. A stack consists of a set of **frames**, each frame containing information about a method call. It includes a local variable array, an operand stack to record intermediate result and a constant pool reference for the called method.

JVM uses the following four registers for each thread, three of which concern the stack:

- program counter
- current frame references
- top of the operand stack
- base of local variable array

Heap

Heap is a shared memory area for all threads of execution. It is used for holding objects (including arrays) after they have been created and while they are used by threads. To conserve memory, objects that are not in use (no variables pointing to it) are removed from thread automatically by the garbage collection process of the JVM. To ensure objects have consistent data when they are accessed by different threads, Java provides a locking mechanism to allow programs manage concurrent accesses to an object.

3.3.2 Class File Format

The Java byte code defines a class file format for each `.class` file that is generated by the `javac` command. It is a structured, machine-independent, binary format that always starts with this fixed “magic number”: `0xCAFEBAE`. The subsequent bytes contain:

- major and minor version numbers of the target JVM
- constant pool
- method table

Example 3.7 Class file of program `HelloWorld`

Listing 3.4 shows a partial Java bytecode that is generated for the HelloWorld program. It was generated by the Java deassembler program (javap), which was invoked as follows:

```
javap -verbose chap2.Hello
```

Listing 3.5 shows a partial constant pool that is generated as part of the Java byte code in Listing 3.4. To ease reading, names of the key sections are highlighted in the listings. Further, the comments inserted for the key elements should help explain what they mean.

Listing 3.4: Java bytecode of HelloWorld

```
Compiled from "Hello.java"
class chap2.Hello
    SourceFile: "Hello.java"
    minor version: 0
    major version: 51
    flags: ACC_SUPER
Constant pool:
const #1 = Class #2; // chap2/Hello
...
public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=1, args_size=1
    0: getstatic    #16  // Field java/lang/System.out:Ljava/io/PrintStream;
    3: ldc          #22  // String Hello, world!
    5: invokevirtual #24 // Method
        java/io/PrintStream.println:(Ljava/lang/String;)V
    8: return
LineNumberTable:
line 5: 0
line 6: 8
LocalVariableTable:
Start  Length  Slot  Name   Signature
      0       9     0  args   [Ljava/lang/String;
...

```

Listing 3.5: Partial constant pool of HelloWorld

```
#1 = Class      #2      // chap2/Hello
#2 = Utf8       chap2/Hello
#3 = Class      #4      // java/lang/Object
#4 = Utf8       java/lang/Object
#5 = Utf8       <init>
```

```

#6 = Utf8      ()V
#7 = Utf8      Code
#8 = Methodref #3.#9    // java/lang/Object."<init>":()V
#9 = NameAndType #5:#6  // "<init>":()V
#10 = Utf8     LineNumberTable
#11 = Utf8     LocalVariableTable
...
#22 = String    #23    // Hello, world!
#23 = Utf8     Hello, world!
#24 = Methodref #25.#27 //
    java/io/PrintStream.println:(Ljava/lang/String;)V
#25 = Class     #26    // java/io/PrintStream
#26 = Utf8     java/io/PrintStream
#27 = NameAndType #28:#29 // println:(Ljava/lang/String;)V
#28 = Utf8     println
#29 = Utf8     (Ljava/lang/String;)V
...

```



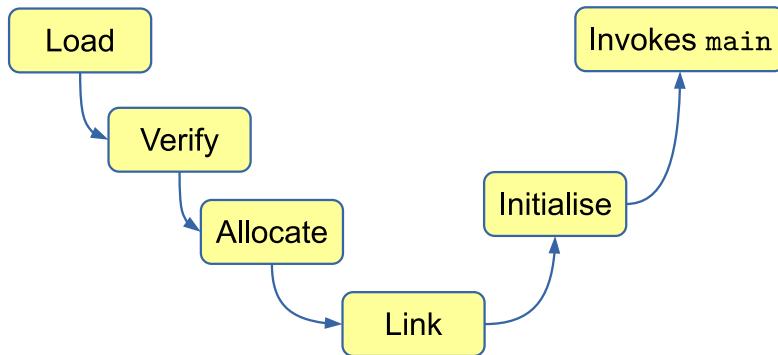
3.4 Java Program Execution

Having understood how the JVM works, let us briefly summarise the process by which the JVM executes a program. For example, what happens when the program `HelloWorld` is executed by the JVM with this command: `java HelloWorld`.

Given that the input class file `C` contains the correctly defined `main` method, the JVM performs the following steps (illustrated in Figure 3.9) to prepare the VM in order to eventually invoke `main` to execute the program:

1. **loads** `C` into memory
2. **verifies** `C` conforms to the class file format
3. **allocates** memory for `C`'s static fields
4. **links** `C` into the run-time library, optionally causing the loading of any additional classes
5. **initialises** `C` by invoking initialisation code for static fields or for `C`
6. **invokes** `main` in a thread.

Method `main` may be invoked with command line arguments. If specified, these arguments are captured in the `String[]` array parameter of the method. For example, the following command invokes `HelloWorld.main` with 3 input strings “say”, “hello”, “world!”:

**Figure 3.9:** Program execution procedure.

```
java HelloWorld say hello world!
```

Listing 3.6 explains how to process the input arguments of method `main`.

Listing 3.6: Processing input arguments

```

1 class HelloWorld {
2     public static void main(String[] args) {
3         for (int i = 0; i < args.length; i++) {
4             System.out.println(args[i]);
5         }
6     }
7 }
```

Note that if lazy loading is applied then the linking step may not load the additional classes immediately. Listing 3.7 shows an example of referencing another class (`Student`) within a program. This class is either lazily loaded by the JVM or loaded immediately as part of linking `HelloWorld` into the run-time library.

Listing 3.7: Referencing another class

```

1 class HelloWorld {
2     public static void main(String[] args) {
3         Student s = null; // a student
4         // initialise s
5         System.out.println("Hello " + s);
6     }
7 }
```

The `main` thread may start additional threads of execution via the `Thread` objects that are created in the program. Listing 3.8 shows an example of creating two `Thread` objects to display different hello messages.

Listing 3.8: Starting multiple threads with class `Thread`

```

1 public class MultithreadHello {
2     public static void main(String[] args) {
3         for (int i = 0; i < 3; i++) { // starts 3 threads
4             Thread t = new Thread(new HelloRunnable());
5             t.start();
6         }
7     }
8 }
```

```

4     final int id = i;
5     new Thread(() -> {
6         System.out.println("Hello thread: " + id);
7     }).start();
8 }
9 }
10 }
```

🌊 Chapter 3 Exercise 🌊

The objective of these exercises is to practise using the parser component of the Java compiler to analyse and write Java programs.

1. Given below is a program named Ex1 that uses a java parser tool, named JavaParser [31], to parse the program text of the class Hello and produce the standard Java source code text. The in-line comments give information about how to use JavaParser.

- (a).** Run this program to produce the source code that is listed immediately below the program.

Note: You need to:

- add to the class path the library file named javaparser-core-3.0.1.jar (provided in the resources/lib folder)
- use import statements to import the necessary classes into your program

- (b).** Change the program text (i.e. value of the variable progText) by removing the semi-colon (';') at the end of the print statement. Rerun the program and record what happens.

- (c).** Change the program text by removing the word “main”. Rerun the program and record what happens.

```

1 import com.github.javaparser.JavaParser;
2 import com.github.javaparser.ast.CompilationUnit;
3 public class Ex1 {
4     public static void main(String[] args) {
5         // program text
6         String progText = "class Hello { "
7             + "public static void main(String[] args) { "
8             + "    System.out.println(\"Hello world!\\");
9             + "}";
10            + "}";
11         // parse the program text
12         CompilationUnit codeUnit = JavaParser.parse(progText);
```

```

13     // obtain the generated source code
14     System.out.println(codeUnit);
15 }
16 }
```

Source code output:

```

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

2. Based on program Ex1, create a new program named Ex2 which reads the same program text from a file named Hello.j, parses it and print the source code to the standard output. Run this program and make sure that it produces the same output as Ex1.

Hint: import the enhanced TextIOPlus class (provided in the library file resources /t3-utils.jar) to read a text file. This class requires the class utils.TextIO (available in the attached source code). For example, to read the file named Hello.j which is located in the same package folder as the class Ex2, you write: TextIOPlus.readTextFileContent(Ex2.class, "Hello.j").

3. Use the enhanced TextIOPlus to write a new program named Ex3 which reads the program text from an URL, parses it and print the source code to the standard output.

The URL of the program text file Hello.j is as follows: <https://sites.google.com/site/lemduc/home/ppl>Hello.j?attredirects=0&d=1>

Hint: use the method TextIOPlus.readTextFromURL.

4. Write a program named ProgWriter (abbr. program writer), which simplifies the process of writing a program using a pre-defined template. This template contains certain fields, whose values are to be specified by the user from the standard input. Take a program that you have written and change it into a template to be used for ProgWriter. You could store the template in a file or simply in a string-typed constant. For example, the program Hello.java would be turned into the following template. This template contains two fields: _ClassName_ and _Greeting_.

```

public class _ClassName_ {
    public static void main(String[] args) {
```

```

        System.out.println(_Getting_);
    }
}

```

5. Provided in the file `t3-utils.jar` is a package named `t3.ast` which contains an implementation of a Java program named `ASTPrinter`. This program uses `JavaParser` to print the abstract syntax tree (AST) of a Java program out on the standard output. This is useful to inspect and study the Java syntax. The header comment of `ASTPrinter` gives very simple instructions on how to use this class in the code. Read this comment and write a program that uses `ASTPrinter` to print the AST of the Java programs that you wrote in the previous tasks.

Compare and contrast the output AST with the relevant grammar rules in JLS. For example, the AST of the program `Hello.java` is printed as follows:

```

CompilationUnit
  ClassOrInterfaceDeclaration
    ClassOrInterfaceDeclaration
      NameExpr: Hello
      MethodDeclaration
        MethodDeclaration
        VoidType: void
        VoidType: void
      NameExpr: main
      Parameter: String[] args
        Parameter: String[] args
        ClassOrInterfaceType: String
        ClassOrInterfaceType: String
        VariableDeclaratorId: args
      BlockStmt
        ExpressionStmt: System.out.println("Hello world!");
        MethodCallExpr: System.out.println("Hello world!")
          FieldAccessExpr: System.out
            NameExpr: System
            NameExpr: out
            NameExpr: println
          StringLiteralExpr: "Hello world!"

```

6. Use the Java disassembler tool (`javap`) to inspect the byte code of one of the programs that you wrote in the previous tasks. The disassembler is located under the `bin` directory of your JDK's installation directory. Run the disassembler three times using the following command-line arguments: `-c`, `-verbose`. Observe and

compare the outputs. For example, if program Ex1 is created in the package named t3, you would type:

```
javap t3.Ex1
```

```
javap -c t3.Ex1
```

```
javap -verbose t3.Ex1
```

Chapter 4

Verifiable Program Development

Objectives

- ✓ Understand the concept of verifiable program and its development.
- ✓ Differentiate and explain the relationship between design specification and implementation.
- ✓ Create design specification of a verifiable procedure.
- ✓ Design a verifiable program from its procedures.
- ✓ Implement a verifiable procedure and verifiable program in Java.

In the previous chapters, we studied the basis of the Java programming language and used it to write programs. In this chapter, we will take a step back to study how to properly develop a program in Java. We aim to achieve *verifiable programs*, which are programs whose behaviours are well-defined and whose implemented codes can later be checked for correctness. Verifiable programs ease understanding, evaluation and maintenance.

To develop verifiable programs, we adapt a stream-lined, structured program development approach, which involves two basic activities: design and coding. The objective of design is to identify and specify the procedural abstractions needed to perform the required tasks. The objective of implementation is then to write code for each abstraction, as a procedure in a target language such as Java, that satisfies its specification. Although we only focus in this chapter on developing procedural programs, the approach can also be applied to the development of object-oriented programs. We will study this in a later chapter.

The rest of the chapter is structured as follows. First, we introduce the basic terminology and define the concept of verifiable procedure – a basic building block of a verifiable program. Next, we explain how to design and implement such verifiable procedure. After this, we define verifiable program and the program development approach. In this, we discuss in detail how to design and implement a verifiable program. We demonstrate the approach with an application example.

4.1 Basic Terminology

Before we start discussing the basic terminology used in this chapter, let us note that this chapter is much influenced by the work by Liskov and Guttag [20]. To ease discussion in this chapter, whenever we reference a specific element of their work, we will use the first author's last name (Liskov) instead of both last names.

4.1.1 Procedure

Definition 4.1. Procedure

A **procedure** is a mapping from (set of) inputs to outputs with possible side-effects.

The inputs, outputs, or both may be empty. A side-effect is a modification to an input.



In Java, procedure is called **method**. To avoid confusion and to keep the terminology consistent, we will use the general term ‘procedure’ throughout this note to refer to Java methods. In the context of this note, we will focus on stand-alone procedures, i.e. procedures that exist in the context of the class, independently from its objects. A stand-alone procedure is invoked directly through the class in which it is defined. It is defined with the keyword `static`. In a later topic, we will study another type of procedure, called *object procedure* (or *operation*). This procedure must be invoked via an object of the class.

Example 4.1 Procedures search and sqrt

```
public static int search(int[] a, int x)
public static float sqrt(float x, float epsilon)
```

The mapping between input and output of the procedure `search` is precisely the mathematical function $\text{IntegerArrays} \times \mathbb{Z} \rightarrow \mathbb{Z}$ where `IntegerArrays` represent all the valid integer arrays and \mathbb{Z} is the set of integers.

Similarly, procedure `sqrt` is defined by the mathematical function $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$. \square

Example 4.2 Procedure sort.

```
public static void sort(int[] a)
```

Procedure `sort` takes an input array `a` and arranges its elements in ascending order. What is interesting here is that this procedure does not return any value so we cannot represent the output in the same way as the two procedures in the Example 4.1. In this case, mapping is a transformation function that modifies the input. \square

4.1.2 Behaviour and Implementation

When we develop a procedure, we have in mind an idea of what it does (i.e. its behaviour) and how to implement it in a target programming language (e.g. Java). Understanding the relationship and distinction between behaviour and implementation is key to effective procedure development.

Behaviour

Behaviour of a procedure is a concise description of a single action or task that it needs to perform. It describes *what* the procedure does, not how it does it. Thus, the behaviour of a procedure needs to be made clear before its implementation can begin. Technically, the behaviour is written using high-level statements (e.g. in natural language or a form of mathematical logic), which are free from details of how the procedure is implemented. Behaviour definition is important not only for the client module that uses the procedure but also for the programmer who implements the procedure. The client can use the procedure based on its behaviour. The programmer writes the code that satisfies the behaviour.

Therefore, it is beneficial to write the behaviour definition as part of the procedure header. Without this, the user would have to read the code to try to work out what the procedure does. This is generally harder than reading the behaviour definition.

Example 4.3 Procedures search and sqrt with behaviour definition.

The comment lines that appear before the header lines of the two procedures search and sqrt below describe their behaviours. We will refer to these descriptions as **behaviour descriptions**. In both examples, the behaviour states properties of the output in terms of the input, without giving details about how the properties are achieved.

```
// if x is in a then return an index where x is stored;
// otherwise return -1
public static int search(int[] a, int x)

// if x is non-negative then return the square-root of x
// within epsilon error
public static float sqrt(float x, float epsilon)
```

For example, search's behaviour states the property `x` in `a` but does not say the actual steps taken to determine this. This allows the programmer to have the flexibility of choosing the suitable algorithm in the implementation. □

Implementation

The **implementation** of a procedure states *how* the procedure performs its behaviour in a specific programming language. In general, different implementations exist for the same behaviour.

Example 4.4 Alternative implementations of procedure search.

The three different implementations of procedure search shown in Listing 4.1 have the same behaviour. The first implementation loops sequentially through the array from the first index. The second implementation uses a similar loop but starts from the last index backward. The third implementation uses two loops: one loops over the even-numbered indices and the other loops over the odd-numbered indices. And so on, we can construct many other implementation variants in this fashion.

Listing 4.1: Alternative implementations of procedure search

```

1 // 1st implementation: forward
2 public static int search1(int[] a, int x) {
3     int i = 0;
4     for (int e : a) {
5         if (e == x) {
6             return i;
7         }
8         i++;
9     }
10 }
11
12 // 2nd implementation: backward
13 public static int search2(int[] a, int x) {
14     for (int i = a.length-1; i >= 0; i--) {
15         if (a[i] == x) {
16             return i;
17         }
18     }
19 }
20
21 // 3rd implementation: splitting
22 public static int search3(int[] a, int x) {
23     for (int i = 0; i < a.length && i % 2 == 0; i++) {
24         if (a[i] == x) {
25             return i;
26         }
27     }
28 }
```

```

29   for (int i = 0; i < a.length && i % 2 == 1; i++) {
30     if (a[i] == x) {
31       return i;
32     }
33   }
34 }
```



Question Which of the implementations shown in Listing 4.1 have the fastest and slowest run times?



4.1.3 Types of Procedure

From the behavioural view point, we can differentiate between two types of procedure [20]: *total* and *partial*. They differ in the assumptions made about the input.

Definition 4.2. Total procedure

A total procedure is a procedure whose behaviour is specified for all values in the input domains.



For example, procedure `search` in Example 4.1 is a total procedure. It does not place any constraints on the input array `a` and the input number `x`. The behaviour is specified for both cases: `x` in `a` and `x` not in `a`.

Definition 4.3. Partial procedure

A partial procedure is a procedure whose behaviour is not specified for some values in the input domains.



For example, procedure `sqrt` in Example 4.1 is a partial procedure. It only specifies the behaviour for the case that the input $x \geq 0$. Nothing is stated in the behaviour description for the case that $x < 0$. In this case, it is generally assumed that the behaviour is undeterministic; that is, it may not return the output that the client expects.

A quick note about the two procedure examples in Example 4.1 is that the totalness property is not explicitly specified. It had to be deduced from the behaviour text. Later in the chapter, we will study a design specification approach that helps make this more explicit in the behaviour.

Each type of procedure has its use in practice but not without trade-offs. Total procedure is safer but requires more effort to construct and generally takes longer to execute. Total procedures handle all input cases and thus are less likely to be broken by an invalid input. Partial procedures can make assumptions (constraints) on the input and

thus have simpler and more efficient code. It is suggested therefore that total procedures should be used in the public contexts, while the partial counterparts be used in more restrictive contexts or when efficiency has a high priority.

4.1.4 Procedural Abstraction

When we define a procedure in a program that performs a task or action, we are effectively defining an abstraction of this task or action. As explained in an earlier chapter, an abstraction is an attempt to conceptualise a problem in which we focus on some details and ignore others. In the case of procedure, the abstraction is based on the view that a task or action involves transforming some input into some output. It is natural to call this abstraction *procedural abstraction* [20], as it serves as the basis for defining a programming procedure.

Definition 4.4. Procedural abstraction.

Procedural abstraction (PA) is an abstraction over a single action or task in the form of a procedure.



Note that PA is not the only programming abstraction that exists. There are other, higher-level programming abstractions, that result from other methods of conceptualisation. One abstraction, which we will study later in this module, is data abstraction. Other abstractions involve some combination of procedural and data abstractions. We will study them in a more advanced module.

Technically, PA is a culmination of two types of abstraction [20]. The first abstraction type, named **abstraction by parameterisation**, is familiar to most programmers because it is the one used in the Java's procedure syntax. In this abstraction, a task/action is viewed in terms of the types of the input and output. The second abstraction type, named **abstraction by specification**, views a task/action in terms of the separation of and the relationship between behaviour and implementation. This is an important type of abstraction but it is often overlooked. It forms the basis for the procedural and program design approach that we will study later in this chapter.

Abstraction by parameterisation. In this abstraction, the identity of the input data values are abstracted by formal parameter's data type. When a procedure is invoked, the parameters are first replaced by (or *bound to*) the arguments and then the procedure's body is evaluated. This abstraction has two important benefits:

- *generalisation*: the procedure is applicable to many input values

- *ease of implementation*: less code is written (one implementation for the entire input type, not the individual input values)

To illustrate, the procedure `search` of Example 4.4 uses two formal parameters `int[]` and `int`. This means that its behaviour is applicable to all integer arrays and integers. This helps avoid the (impossible) task of implementing the same behaviour for each individual pairing of array and integer.

Abstraction by specification. This abstraction refers directly the relationship between behaviour and implementation discussed earlier in Section 4.1.2. It is clear that the behaviour of a procedure is an abstraction of its implementation. At the behavioural level, the relevant detail is *what* rather than how the procedure performs a task. This is called **abstraction by specification**, because the behaviour is specified explicitly for the procedure.

Liskov and Guttag [20] state two important benefits that this abstraction brings to program design: *locality* and *modifiability*. **Locality** is the extent to which an abstraction (procedure in this case) can be implemented independently from other abstractions. On the other hand, **modifiability** is the extent that implementation changes can be applied to an abstraction without impacting other abstractions that use it.

Example 4.5 Procedural abstraction

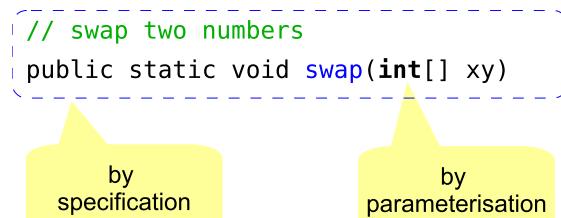


Figure 4.1: Procedural abstraction types of the procedure `swap`.

To illustrate both abstraction types of PA, we introduce in Figure 4.1 another procedure called `swap`. We will use this procedure later in this chapter when we discuss our design approach. From the abstraction-by-parameterisation view point, an input parameter typed `int []` is used as an abstraction over all the two-element integer arrays (e.g. `[1,2]`, `[3,10]`, ...) that are the input data of the procedure. From the abstraction-by-specification view point, the behaviour description in the header comment describes the behaviour of the procedure (what it does). It states the swap action, which in this context means to swap the positions of the elements of the input array. However, it does not give details of how this swapping can be carried out. One way is to use a temporary array, another way is to use a temporary variable, and many other approaches similar to these. □

4.1.5 Procedure Invocation

It is a statement that call (or invoke) another procedure. The Java syntax for this was examined in a previous chapter. Its basic form is as follows:

`<class-name>.<procedure-name>(<argument-list>)` where:

- `<class-name>`: the name of the enclosing class (may be omitted if in the same class)
- `<procedure-name>`: the name of the called procedure
- `<argument-list>`: the values that are passed into the procedure as input. These values must match the data types of the parameters.

When the invocation statement is placed in the body of another procedure. This procedure is named the invoking or **calling procedure**. The procedure being invoked is named the invoked or **called procedure**. If the called and calling procedures are defined in the same class then the class name may be omitted from the invocation statement.

Example 4.6 Procedural invocation

Listing 4.2: Procedural invocations in `Sum.main`.

```

1 public static void main(String[] args) {
2     float s = 0;
3     float[] r = getNumbers();
4     s = sum(r);
5     TextIO.putln("sum = " + s);
6 }
```

Listing 4.2 shows several examples of procedural invocations in the procedure `Sum.main`. This procedure invokes three other procedures: `getNumbers`, `sum` and `TextIO.putln`. □

4.2 Verifiable Procedure Design

In the general software development process [30, 33], verification is performed at the end of each phase to check that its deliverables satisfy the requirements defined in deliverables of the previous phase. In the coding phase, verification is used to verify that a module's code satisfies its design specification. It is therefore important that the specification is properly defined and accessible to the programmer. In this section, we will discuss how to write a well-defined specification for the procedures that make up a program. In Section 4.4, we will generalise this for program design specification.

Why do we care about well-defined behaviour? Why can not we just write a simple comment to describe the behaviour as shown in the procedure `swap` of Example 4.5. In

that example, is the behaviour description “swap two numbers” clear enough for the programmer to understand to the extent that would prevent them from making logic-related mistakes in coding it? There are two limitations of that behaviour description:

1. What conditions (if any) do we need to impose on the parameter `xy` in order for the behaviour to be valid?
2. What exactly does “swap two numbers” mean in terms of the parameter `xy`?

A well-defined behaviour description is one that should address these. This description, which is called **design specification**, makes more explicit and precise the behaviour so that the client can correctly understand and rely on the behaviour. On the other hand, the programmer is able to correctly implement the procedure. The correctness of the written code can be verified with regards to (*w.r.t*) the specification. In this book, we call a procedure that has a well-defined specification a *verifiable procedure*.

Definition 4.5. Verifiable procedure

Verifiable procedure is a procedure that has a well-defined design specification.



The design specification of a verifiable procedure forms a formal *contract* [20] between the client and the programmer. The client agrees to use the specification as written, while the programmer agrees to provide a correct implementation of the specification. This separation of concerns helps enable modular program development. To ease explanation in the rest of this chapter, the phrase “*design specification*” will mean a well-defined specification.

4.3 Design Specification Language

To write a design specification requires a **design specification language**. It is important for this language to be understandable by both the client (typically a non-technical person) and the programmer. To achieve this, this book adapted the Liskov’s semi-structured specification language of [20]. This language combines concise, semi-structured NL statements with logical statements (for preciseness, where necessary). Unlike the Liskov’s approach, however, this book uses the block comment format to write the specification. This format is supported by most modern OOPs (such as Java, C#). Further, this book uses text annotations within the comment to more explicitly mark the specification sections.

4.3.1 Block Comment Format

In Java, the block comment used as the header comment of a procedure can be used by the javadoc tool of the language to automatically generate a HTML-based, technical document of the procedure. The block comment format is “(`/** */`)”. The block comment also supports HTML tags (such as `
`, `<p>`, `<pre>`, and the special tag `<tt>` for writing technical terms (e.g. formulas). In addition, we can use text annotations, which have the prefix ‘@’, to define the specification elements. We will discuss next how to use these annotations to structure the behaviour description.

4.3.2 Behaviour Specification Structure

This book adopts the Liskov’s specification structure [20] but proposes to use text annotations to explicitly define this structure. This structure consists of three core components: pre-conditions, side-effects, and post-conditions.

Pre-conditions state the conditions (if any) that must hold true *before* the behaviour can be invoked. It typically describes conditions (or *constraints*) on the input parameters. Partial procedures (see Section 4.1.3) are always specified with some pre-conditions. Pre-conditions are written in a clause named `REQUIRES`, using annotation `@requires`.

Side-effects (if any) are written in a clause named `MODIFIES`, using the annotation `@modifies`. It simply lists names of the parameter(s) whose values are modified by the procedure.

Post-conditions state the conditions that must hold true *after* the behaviour is invoked. They are written in a clause named `EFFECTS`, using the annotation `@effects`. The content should state the transformation of the input into output.

In addition to the core annotations above, we introduce two other annotations: `@pseudocode` and `@overview`. It is observed that in cases where the behaviour is either very complex or very domain-specific, a specific algorithm is additionally provided for a procedure. This has the effect of further restricting the set of suitable implementations of the procedure. For this case, annotation `@pseudocode` is be used to write the algorithm.

Unlike the previous annotations, which are applied to procedure, annotation `@overview` is used for class. It is included as part of the class’s header comment to describe the purpose of the class.

Example 4.7 Design specification of procedure `swap`.

Listing 4.3: Design specification of procedure `swap`

```

1 /**
2 * Swap two numbers

```

```

3 * @requires xy != null /\ xy.length=2
4 * @modifies xy
5 * @effects xy = [xy_0[1], xy_0[0]]
6 */
7 public static void swap(int[] xy)

```

Listing 4.3 shows the design specification of procedure `swap`. This specification overcomes the limitations discussed earlier about the unstructured description in Example 4.5. First, the `@requires` clause states the pre-conditions on the parameter `xy`, that it must be defined and contains exactly two elements. Second, the `@modifies` clause states clearly that `xy` is modified and the modification is specified precisely in the `@effects` clause.

Note that in this case, the pre- and post-conditions can be written directly and precisely in logical notation. We will discuss this notation in the next section. The logical statement in the `@effects` means that the value of `xy` after invoking the behaviour is an array that is constructed from two elements: `xy_0[1]` and `xy_0[0]`. Here, `xy_0` refers to the value of `xy` at entry into the procedure (i.e. before the behaviour is executed). What this means, therefore, is that the elements of `xy` are swapped for the two elements that were in it initially. \square

Example 4.8 Design specification of procedure `intMax`.

Listing 4.4: Procedure `intMax`

```

1 /**
2 * Determines the greater of two numbers
3 * @effects <pre>
4 *   (result = x0 \vee result = y0) /\ 
5 *   (result >= x0 /\ result >= y0)
6 * </pre>
7 */
8 public static int intMax(int x, int y)

```

Listing 4.4 shows the design specification of the procedure `intMax`. This specification has no pre-condition and side-effects and so the annotations `@requires` and `@modifies` are omitted. The post-condition describes exactly what the max value is: one of the two numbers which is greater than or equal to both of them.

To illustrate the use of HTML tags in the Java-style comment, we include in this specification the HTML tag `<pre>`, which wraps around the content of `@effects`. This means that this content is assumed to be pre-formatted and is presented raw to the user. This tag is useful for technical content that may contain special characters. \square

Example 4.9 Behaviour Specification: `Sum.main`, `sum`.

In order to prepare for program design specification that will be discussed later, we introduce here the design specifications of two procedures of a program named Sum. The design specifications of two procedures are clear enough to understand what they do without explain what the program does.

Listing 4.5: Two procedures of program Sum

```

1  /**
2   * the main procedure
3   * @effects
4   * obtain some real numbers from the user
5   * {@link #sum(float[])}: compute the sum
6   *      of the numbers
7   * print the sum
8  */
9 public static void main(String[] args)
10
11 /**
12 * Determines the sum of an array of real numbers.
13 * @requires <tt>a is not null</tt>
14 * @effects <pre>
15 *      return the sum of the array elements, i.e.
16 *      result = a[0]+...+a[a.length-1]
17 *      </pre>
18 */
19 public static float sum(float[] a)
```

Listing 4.5 shows the design specifications of two procedures of program Sum: `main` and `sum`. Procedure `sum`'s pre-conditions state that the input array must be defined. This procedure has no side-effects. The post-conditions state what the sum means: the arithmetic sum of elements of the input array.

In both specifications, short natural language statements are used. Further, a notation for procedural invocation is used in the statement `{@link #sum(float[])}`. Annotation `@link` tells the javadoc tool to generate a clickable link to the invoked procedure. This is useful for users to easily navigate between procedures in the program. We will discuss these elements in more detail in the next section. □

4.3.3 Language Constructs

In this section, we summarise the design specification language constructs that are used in this book. Some of these constructs were adapted from Java and are expressed in a PL-independent manner. We define the constructs informally using natural language.

- logical notation as shown in Table 6.2.
- reserved terms for 5 specification components: @requires, @modifies, @effects, @pseudocode, @overview
- single and block comments
- primitive and array types
- null type (same meaning as in Java)
- statements end with the next line character
- statement block is organised using indentation (no curly brackets)
- core statements:
 - variable declaration and assignment (adapted from Java)
 - conditional and loop (adapted from Java)
 - read: read some data from some input
 - print: display some data to the standard output
 - return: return some data as output
- array operations:
 - add x to a: add x to the next index position in a
 - put x in a: put x in any index in array a
 - delete x in a: set the first item matching x in a to a pre-defined constant used to denote the discarded value
- operators: eq (==), noteq (!=), lt (<), gt (>), etc.
- short natural language statements are allowed
- procedural invocation (as specified in the javadoc format):


```
{@link <class_name>#<method_name>(<parameter_types_list>)}
```

 - <class_name>: name of the class that contains the called procedure (omitted if it is the same class)
 - <method_name>: name of the called procedure
 - <parameter_types_list>: list of the parameters of the called procedure but without the parameter names

Table 6.2 lists the logical notation and their textual form used in the specification.

4.3.4 What Makes a Good Specification?

Liskov and Guttag [20] states three main criteria for judging the quality of a design specification: *restrictiveness*, *generality* and *clarity*. Table 4.2 summarises these three criteria and give examples to illustrate. It can be drawn from this table that an interesting property of a good specification, which satisfies all three criteria, is that it manages to

Table 4.1: Logical notation

Logical symbols	Textual form	Description/Example
\wedge	$\wedge\backslash$	conjunction (logical AND)
\vee	$\vee\backslash$	disjunction (logical OR)
\rightarrow	\rightarrow	implication (if/then)
\leftrightarrow	\leftrightarrow	if and only if
		e.g. for all x : nat. $p(x)$
\forall	for all	means $p(x)$ holds for all natural number x (nat. denotes the set of all natural numbers)
		e.g. exists x : nat. $p(x)$
\exists	exists	means $p(x)$ is true for at least one natural number x

balance between the opposing factors. The first two criteria are opposing because one restricts while the other widens the specification scope. Putting these criteria together basically means that the specification should aim to cover most valid implementations and no invalid ones. The third criteria has no tensions with the other two and is generally applicable to all specifications. However, this criteria itself consists in two opposing subcriteria. These subcriteria mean that the specification should aim to be short and clear for both client and programmer.

Table 4.2: Design specification quality criteria

Criteria	Description	Opposing
Restrictiveness	extent to which unsatisfactory implementations are avoided. <i>Example:</i> includes @requires when necessary.	Generality
Generality	extent to which satisfactory implementations are included. <i>Example:</i> use definitional-style (<i>what</i>) description.	Restrictiveness
Clarity	extent to which statements are clear to both client and programmer. Balance between <i>conciseness</i> and <i>redundancy</i> . Conciseness means short and clear, while redundancy means to use alternative (more user friendly) statements where possible. <i>Example:</i> Conciseness: use high-level conditional and loop-style statements; Redundancy: use ‘i.e.’ to give precise technical meaning and ‘e.g.’ to give an example	conciseness vs. redundancy

Example 4.10 Procedure swap

```


    /**
     * Swap two numbers
     * @requires xy != null /\ xy.length=2
     * @modifies xy
     * @effects xy = [xy_0[1], xy_0[0]]
     */
    public static void swap(int[] xy)


Restrictive      General (definitional style)      Concise



```


 /**
 * Swap two numbers
 * @requires xy != null /\ xy.length=2
 * @modifies xy
 * @effects xy = [xy_0[1], xy_0[0]]
 * e.g. xy=[1,2] /\ swap(xy)=[2,1]
 */
 public static void swap(int[] xy)


```


```

Figure 4.2: Applying the specification criteria to the procedure `swap`.

Figure 4.2 illustrates using the three criteria to analyse the design specification of the procedure `swap` (see Listing 4.3). The comment boxes in the figure should be self-explanatory. Note, in particular, that the specification includes an example statement ('e.g. '), which appears at the end of the `@effects` clause. This is useful for non-technical user (the client) to understand the notation used in the clause for computing `xy`. \square

4.4 Verifiable Program Design

In this section, we discuss how to design a program from its procedures. Similar to procedures, the aim is to achieve *verifiable programs*. Let's first discuss a number of background terms that we will need for the design.

4.4.1 Preliminaries

At the program level, the design makes use of the following terms:

- Procedural program
- Program behaviour
- Program structure
- Functional procedure
- Program and functional classes

Definition 4.6. Procedural program

A *procedural program* is a set of procedures, exactly one of which is the procedure `main`, and a set of procedure invocations. Procedure `main` is used to start the program.



Definition 4.7. Program behaviour

Program behaviour is defined by the behaviour of procedure `main`, which in turn is collectively defined from the procedures' that are invoked by `main`.



Because of its special role, procedure `main` has a fixed syntax in most OOPs. Listing 4.6 shows two very similar syntax of this procedure that are used in Java and C#. In Java, procedure `main` must be defined using the fixed syntax shown in the listing. The C#'s syntax requires that the method is named `Main` but is a bit more flexible in that the method's modifier needs not be `public` and that the method can return an `int` value.

Listing 4.6: Syntax of procedure `main` in Java and C#

```

1 // Java's syntax
2 public static void main(String[] args)
3 // C#'s syntax
4 static void Main(String[] args)
5 static int Main(String[] args)
```

In program development, an important step often performed before writing the detailed design specification is analysing the program requirement to construct its structure. This structure shows the procedures and their invocation relationships.

Definition 4.8. Program structure

Program structure is a directed, binary graph that has the following properties:

- nodes represent procedures.
- edges represent procedure invocations.
- edges originating from the same node arranged from left to right in invocation order.
- a connected subgraph whose edges represent the invocations owned by one higher-level procedure is called the **procedure graph**.
- a special edge (named `return`) represents the return statement of a procedure. This edge connects the node that precedes `return` to a side of the procedure context.

To ease comprehension, nodes are arranged in the invocation order. Each procedure graph is bounded by a dashed rectangle. This rectangle represents the **procedure context** (which is the scope of the procedure's body).



Note that procedure graph and its context are necessary to make clear the boundary of each procedure. Without them, it would be possible, for instance, to mistake the program graph for being owned by just the method `main`. To visually draw the program

structure, this book adapts the flowchart symbols¹ [21] and introduces two new symbols. The first symbol is a dashed box for procedure context and the second symbol is a dashed arrow that represents the returned value from a called procedure. Note that for illustration purpose, we embed the context of a called procedure directly in the program graph of the calling procedure. In the general case where a procedure may be invoked by several other procedures, its context is drawn as a separate graph and is linked to the program graph of the calling procedures.

Example 4.11 Program Sum

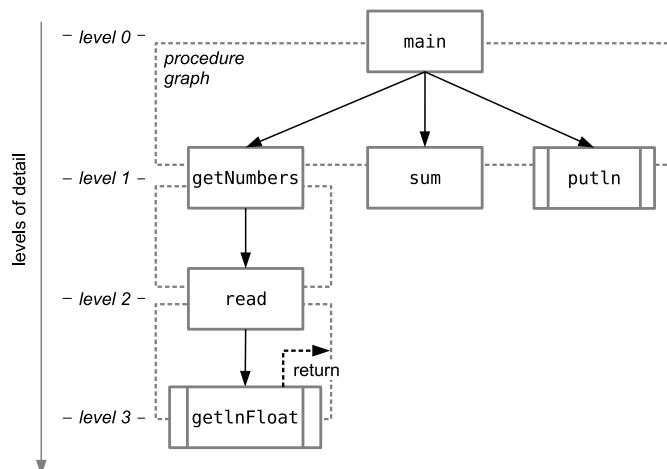


Figure 4.3: The structure of program Sum.

Figure 4.3 graphically shows the structure of a program named `Sum`. We will study this program design and its code in more detail later in the chapter. What is of interest here is the result of analysing the program requirement to build the overall program structure. In this example, we will informally describe the procedures in natural language. As shown in the figure, the program structure consists in three procedure graphs, one owned by `main`, the other two are owned by `getNumbers` and `read` (respectively). Nodes are arranged from top down, in invocation order. This arrangement is intuitive in that it shows the increasing level of detail. Procedure `main` is placed at the top (level 0). Three procedures invoked by `main` are positioned at level 1. There are three directed edges connecting `main` to these procedures. Procedure `getNumbers` obtains some real numbers from the user via the standard input. Procedure `sum` computes the sum of these numbers. Procedure `putIn` is then invoked to display the result on the standard output. This last procedure is defined in an external library named `TextIO` and is thus represented by a different symbol (a rectangle with 2 vertical lines on two sides).

At levels 2 and 3 of the program structure are two procedures that are used by `getNumbers` to perform its task. Procedure `read` (invoked directly by `getNumbers`)

¹<https://en.wikipedia.org/wiki/Flowchart>

reads a real number from the user. Procedure `getInFloat` (invoked by `read`) actually obtains the real number from the standard input. It is defined in the library `TextIO`. Since procedure `read` returns a value and it only invokes procedure `getInFloat`, this latter node is connected to a border of `read`'s context via a return edge. □

Definition 4.9. Functional procedure

A functional procedure is a procedure other than the procedure `main`. Functional procedure thus performs the domain-specific tasks. A internal functional procedure is one that is defined in the program, an external functional procedure is defined in an external library.



A procedure is defined in a class, which gives rise to two terms used for class.

Definition 4.10. Functional class

Functional class is a class that only contains internal functional procedures.



Note that in an OOPL, only one procedure `main` can be used to execute a program. The class that contains this procedure is called the *program class*.

Definition 4.11. Program class

Program class is a class that contains procedure `main` and possibly other internal functional procedures.



Example 4.12 Multi-class program Sum.

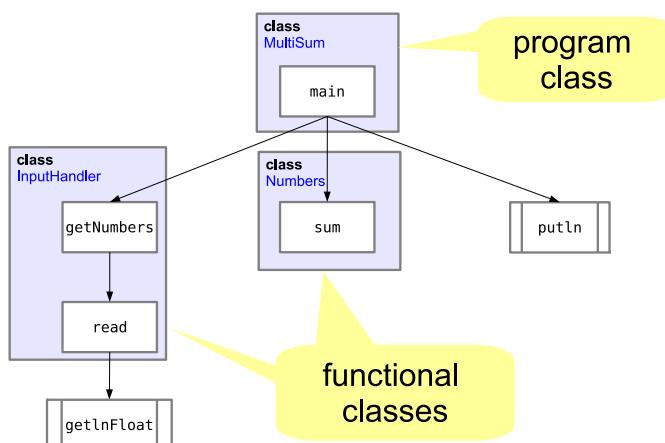


Figure 4.4: The structure of program Sum with multiple classes.

Figure 4.4 shows another structure of program Sum that contains three classes. Each class includes a group of related internal procedures. For example, class `MultiSum` is the program class. Two other classes (`InputHandler` and `Numbers`) are functional classes, each of which contains a group of procedures that realise the class's behaviour. □

Finally, we introduce the term *verifiable program*.

Definition 4.12. Verifiable program.

Verifiable program is a procedural program all of whose procedures are verifiable. 

4.4.2 Structured Program Design

Structured program design originates from structured programming, which whose benefits were recognised by Dijkstra [7, 8] in the early days of high-level programming. The basis of structured program is the divide-and-conquer principle, which is an ancient method used to solve complex problems. The idea is to break a problem into smaller subproblems that are easier to solve and whose solutions can be combined to form the solution of the original one. In a more modern tone, Liskov and Guttag [20] states that *functional decomposition* is the essence of structured program design. The aim is to gradually divide a program into procedures in such a way that combining these procedures produce the behaviour of the original program. This decomposition process produces in a program structure (see Definition 4.8).

Definition 4.13. Structured program design

Conceptually, **structured program design method** proceeds as follow:

1. Determine program behaviour from the requirement
2. Decompose it into smaller procedures at the next level of detail
3. Repeat step 2 for each newly created procedure, until no further decomposition is necessary (i.e. the desired level of detail has been achieved). 

An example of structured program design is presented in our explanation of the program structure in Example 4.11. Functional decomposition was clearly used in that example to identify the procedures and their invocations.

The next definition gives details of a design technique that realises the method described above in an OOPL. To ease discussion, we will assume a single-class design for most program examples. The transformation from single-class to multi-class design is straightforward, as illustrated in Example 4.12. It is the simple matter of grouping procedures whose behaviours are related according to some high-level categories and identifying suitable classes to contain the procedure groups.

Definition 4.14. Program design technique

A *structured program design technique* for OOPL proceeds with three steps:

1. Specify the program class.
2. Specify procedure main.

3. Specify the functional procedures.



The design steps are defined based on the OOPL program structure discussed in Definition 4.8. The subsequent sections will explain these steps in detail.

4.4.3 Specifying the Program Class

Class specification consists in the header and header comment that describes the purpose of the class. A typical **class header** is defined as follows:

```
public class <class-name>
```

where `class-name` is an informative name that is derived from the problem description. If the class is not to be accessed directly from other components of the program then the modifier `public` may be removed or changed to `private`. As a convention, every word in the class name starts with an upper case character; e.g. `ArithmeticSum`, `SwapTwoNumbers`, and so on.

The **class header comment** should state the program purpose, which is taken from the problem description. It is written with the annotation `@overview`. Other annotations (e.g. `@author`) may also be used to give more information about the class.

Example 4.13 Program class specification

Listing 4.7 gives the specification of the program class `Sum`. The structure of this program was described in Example 4.11. In this example, the program name is rather obvious. The `@overview` annotation of the header comment describes the expected program behaviour of this program.

Listing 4.7: Program class Sum

```

1 /**
2 * @overview a program that computes the sum of real numbers
3 * obtained from the user via standard input
4 */
5 public class Sum {
6     //
7 }
```



4.4.4 Specifying Procedure `main`

The next step is to specify the entry point of the program. The procedure's header only needs to observe the pre-defined syntax of the OOPL. The behaviour description

(written in the header comment) needs to summarise the program behaviour in a slightly more detail than that given in the class header comment. Since procedure `main` typically invokes a number of non-functional procedures to perform more specific, non-trivial tasks, its behaviour description often includes procedural invocations for those procedures.

Example 4.14 Procedure `main` specification.

Listing 4.8 lists the specification of the procedure `Sum.main`. The behavioural description contains an NL statement about the behaviour and an ‘i.e.’ paragraph, which lists the three steps that are performed. Each step contains a procedure invocation and a description. The description of each invocation is simply obtained from the behaviour description of the invoked procedure.

Listing 4.8: Procedure `Sum.main`

```

1 /**
2 * @effects
3 *   computes the arithmetic sum of real numbers
4 *   obtained from the user via standard input.
5 * i.e.
6 *   {@link #getNumbers():} obtain some real numbers from the user
7 *   {@link #sum(float[]):} compute the sum of the numbers
8 *   {@link TextIO#putln():} print the sum
9 */
10 public static void main(String[] args)

```



4.4.5 Specifying Functional Procedures

The header of a functional procedure needs to observe the OOPL’s syntax for stand-alone procedures. In particular, it must use the keyword `static` and should be specified with an access modifier (`public`, `private` or `protected`) suitable for its context in the program. If the method is not expected to return a value then its return type needs to be set to `void`. As a naming convention, all but the first word of the procedure name should start with an upper-case character.

Each functional procedure performs a specific task and thus its behaviour should be defined based on what this task is. The specification language defined in Section 4.3 is used to write the behaviour description. In particular, the three quality criteria (restrictiveness, generality and clarity) should be observed.

Example 4.15 Functional procedure specification.

Listing 4.9 lists the specifications of three functional procedures of program Sum. Procedures `getNumbers` and `sum` are invoked by `main`, while procedure `read` is invoked by `getNumbers`. The specification texts in the listing are self-explanatory.

Listing 4.9: Functional procedures of program Sum

```

1 /**
2  * Determine the sum of array of real numbers.
3  * @requires a is not null
4  * @effects return the sum of a,
5  *          i.e. result = a[0]+...+a[a.length-1]
6 */
7 private static float sum(float[] a)
8
9 /**
10 * Obtain an array of real numbers from the standard input.
11 * @effects
12 * prompt user to enter n = how many numbers
13 * initialise a = float array of size n
14 * while a is not filled
15 *   f = {@link #read():} prompt user to enter a number
16 *   add f to a
17 * return a
18 */
19 private static float[] getNumbers()
20
21 /**
22 * Read a real number from the standard input.
23 * @effects
24 * do
25 *   f = {@link TextIO#getlnFloat():}
26 *         prompt user to enter a number
27 *   while f is invalid
28 *   return f
29 */
30 private static float read()

```



4.5 Implementation in Java

Once the detailed design has been specified, we can proceed to coding the design in the target OOPL. Strictly in the software development process [30], coding is performed

as part of the implementation phase. However, in programming context (e.g. [20]), the two terms are often used interchangeably. Since this book's scope is in the programming context, all references to the term "implementation" mean the same as coding.

Although Java is selected as the target OOPL, the approach described here would be applied equally well to C# and other OOPLs.

Definition 4.15. Coding method

Given the design specification of a program, the steps to code this design mirror those of program design presented in Definition 4.14. Specifically, coding proceeds as follows:

1. *Code the program class.*
2. *Code procedure main.*
3. *Code each functional procedure in the invocation order.*
4. *If multi-class design is used, repeat steps 1 – 2 for each functional class.*



Note that coding steps 2 and 3 can also be performed in the reverse invocation order. The order of these two steps determine the *coding strategy*. The strategy specified in Definition 4.15 is the *top-down strategy*, which conforms to the top-down program structure. In contrast, the *bottom-up strategy* involves coding the procedures in reverse invocation order. Although both strategies arrive at the same outcome, there is a key point of difference. That is, in the top-down strategy, the invoked procedures are initially coded as *stubs* [20]. A **stub procedure** has a dummy body and is used to satisfy the invocation requirement of a calling procedure. The body of a stub will eventually be written when the coding process gets to it.

Example 4.16 Program Sum coding

Listing 4.10 shows the Java code of program Sum. The design specifications of all but procedure `sum` are omitted. Refer to Listing 4.7 for the complete design specification of this program. Procedure `sum`, in particular, uses a `while` loop to iterate over the input array elements and add them to the running total (variable `s`). Alternatively, we could easily implement the same loop using the `for` statement. Although the code of this procedure is perhaps one of the most basic pieces of code that a beginning programmer would write, it quite clearly demonstrates the advantage having the design specification. First, the code is longer than the `@effects` clause in the specification. This clause simply writes the sum using the fundamental addition rule. The client can take for granted that this rule is correct. Second, it is not immediately clear whether the code is correct. Formally proving the correctness of a loop requires using logical induction, which is not trivial!

Listing 4.10: The Java code of program Sum

```

1 package chap4.implement;
2 import utils.TextIO;
3 /**
4 * @overview A program that computes the sum of real numbers
5 * that are obtained from the user via the standard input
6 * @author ducmle
7 */
8 public class Sum {
9 /**
10 * (omitted)
11 */
12 public static void main(String[] args) {
13     float s = 0;
14     float[] r = getNumbers();
15     s = sum(r);
16     TextIO.putln("sum = " + s);
17 }
18
19 /**
20 * Determine the sum of array of real numbers
21 * @requires a is not null
22 * @effects return the sum of a's elements,
23 * i.e. result = a[0] + ... + a[a.length-1]
24 */
25 private static float sum(float[] a) {
26     int n = 0;
27     float s = 0;
28     while (n < a.length) {
29         s = s + a[n];
30         n++;
31     }
32     return s;
33 }
34
35 /**
36 * (omitted)
37 */
38 private static float[] getNumbers() {
39     // ask user for how many numbers she needs
40     // prompt the user for the number of numbers needed
41     TextIO.putln("How many numbers do you need?");
42     int n = TextIO.getlnInt();

```

```

43     while (n < 0) {
44         TextIO.putln("Invalid number: " + n + ". Please reenter.");
45         n = TextIO.getlnInt();
46     }
47     // initialise an array whose size is the specified number
48     float[] a = new float[n];
49     // fill a with numbers
50     float f;
51     for (int i = 0; i < n; i++) {
52         f = read();
53         a[i] = f;
54     }
55     return a;
56 }
57
58 /**
59 * (omitted)
60 */
61 private static float read() {
62     TextIO.putln("Enter next real number:");
63     float num = TextIO.getlnFloat();
64     return num;
65 }
66 }
```



4.6 Application Example: Coffee Tin Game

In this section, we apply the program development method defined in this chapter to develop a non-trivial program named `CoffeeTinGame`. This program is adapted from Eisenbach et al. [9], which simulates a fun game that demonstrates the loop construct.

4.6.1 Problem definition

The coffee tin game involves a tin full of two kinds of coffee bean: Blue Mountain (code name blue or simply B) and Green Valley (code name green or simply G). The game proceeds as shown in the high-level pseudocode in Listing 4.11. It basically involves a sequence of game moves, each of which takes out some beans from the tin according to a set of rules. Because of these rules, it is possible to show that there is always one bean left in the tin and that its color can be determined regardless of the game

move. The objective of program `CoffeeTinGame`, therefore, is to execute the game and to determine its outcome.

Listing 4.11: A high-level algorithm of the Coffee tin game.

```
// what is the colour of the final bean?  
while at least two beans in tin do  
    take out any two beans  
    if they are the same colour  
        throw them both away  
        put a Blue Mountain bean back in  
    else  
        throw away the blue one  
        put the green one back
```

4.6.2 Analysis

Let us analyse the game moves to show how the color of the last bean is determined. The tin's content (t) at any given time is represented by the numbers of blues (m) and greens (n): $t = m + n$. A game move is determined by colours of the two beans taken out, which can be one of the followings: BB, BG, GG. These game moves change the tin's content as follows:

- BB: $m + n \rightarrow (m - 1) + n$
- BG: $m + n \rightarrow (m - 1) + n$
- GG: $m + n \rightarrow (m + 1) + (n - 2)$



Question What do you notice about n and the total number of beans ($m + n$) ?

Property of the last bean

Given the game moves above, the colour of the last bean can be determined from initial beans in the tin, regardless of the number of game moves. Specifically, it can be observed that:

- m is either increased or decreased by one
- n is either unchanged or decreased by 2, i.e. the parity of the number of green beans ($\text{mod}(n, 2)$), is unchanged
- two beans are required to perform the move but t is decreased by one, i.e. the game is stopped when there is one bean left in the tin

The last bean can be either G or B, but which is it? If it is green then it must be that $\text{mod}(n, 2) = 1$; otherwise $\text{mod}(n, 2) = 0$. Thus, the **last bean property** (its color) can be formulated based on the initial number of greens (n_0) as follows:

$$\text{last bean} = \begin{cases} \text{G}, & \text{if } \text{mod}(n_0, 2) = 1 \\ \text{B}, & \text{if } \text{mod}(n_0, 2) = 0 \end{cases}$$

Program Structure

Figure 4.5 shows the structure of program CoffeeTinGame. It has 3 procedure graphs whose contexts are drawn in the figure. The first graph is owned by procedure `main`, the second is owned by `tinGame` and the third is own by `takeTwo`.

The procedure graph of `tinGame` contains a new symbol (\diamond), which we did not discuss earlier in this chapter. This symbol represents **decision node** [21], which has one incoming edge and two or more outgoing edges. The outgoing edges represent different decision outcomes. There are two decision nodes in the graph. The first node appears after the invocation of procedure `hasAtLeastTwoBeans`. It has two outcomes Y and N. If Y (i.e. tin has at least 2 beans) then `takeTwo` is invoked to obtain two beans from tin. If N then `anyBean` is invoked, which is followed by a return statement to return any bean left in tin.

The second decision node appears after `takeTwo`'s invocation. It has 3 outcomes, representing the 3 cases of the beans taken out. All three cases cause an invocation of `putIn` (using a different input each time). This invocation is to put one bean back into the tin. After this, execution repeats at the invocation of `hasAtLeastTwoBeans`.

Last but not least, the procedure graph of `takeTwo` contains an invocation to procedure `takeOne`. The reason we introduce this procedure is because it is performed twice per game move to get 2 beans needed for the move.

4.6.3 Design

In this step, the design specifications of each procedure and the entire program are written. Listing 4.12 shows the brief design specification of program CoffeeTinGame. It lists the procedure headers but omits the behaviour descriptions. These descriptions will be presented for each procedure in the subsections below.

Listing 4.12: Brief design specification of program CoffeeTinGame

```

1 package chap4.design;
2 /**
3 * @overview A program that performs the coffee tin game on a
4 *   tin of beans and display result on the standard output.

```

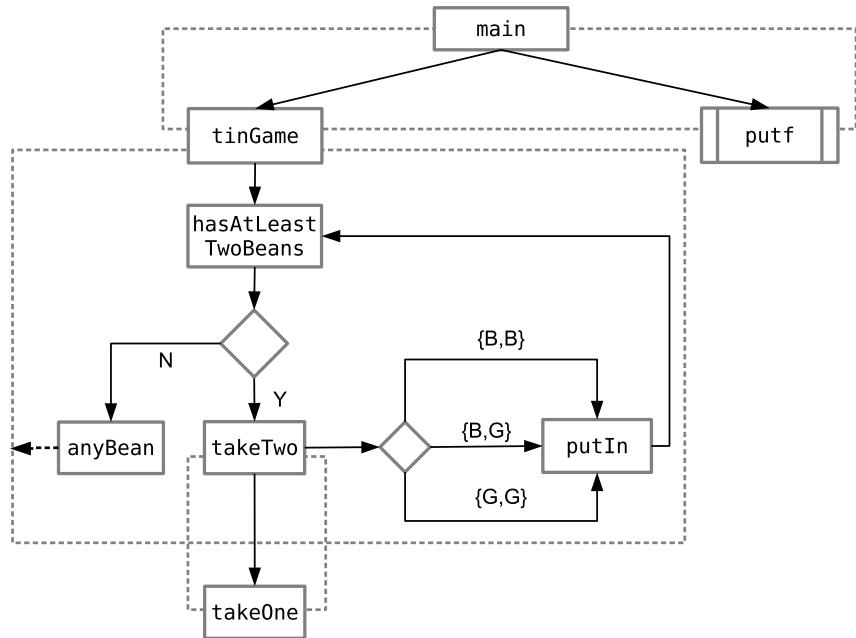


Figure 4.5: Structure of program CoffeeTinGame.

```

5  */
6 public class CoffeeTinGame {
7  /** (omitted) */
8  public static void main(String[] args)
9
10 /**
11  private static char tinGame(char[] tin)
12
13 /**
14  private static boolean hasAtLeastTwoBeans(char[] tin)
15
16 /**
17  private static char[] takeTwo(char[] tin)
18
19 /**
20  private static char takeOne(char[] tin)
21
22 /**
23  private static void putIn(char[] tin, char bean)
24
25 /**
26  private static char anyBean(char[] tin)
27 }

```

Procedure main

Listing 4.13: Design specification of procedure main

```

1  /**
2   * @effects
3   *   initialise a coffee tin
4   *   {@link TextIO#printf}: display tin content
5   *   {@link #tinGame(char[]}[]): perform coffee tin game on tin
6   *   {@link TextIO#printf}: display tin content
7   *   if last bean is correct
8   *     {@link TextIO#printf}: display last bean
9   *   else
10  *     {@link TextIO#printf}: display error message
11 */
12 public static void main(String[] args)

```

The basic tasks of procedure `main` are to initialise a tin of coffee beans, call `tinGame` to perform the game and then call `TextIO.printf` to display the result. In addition to these, procedure `main` contains two calls to `TextIO.printf` to display the tin content before and after the game. This is to let the user see the effect of the game on tin. Further, it uses the property of the last bean to check the value returned from `tinGame`. Although this task is not essential to the program, it is used here to test the implementation of the procedure `tinGame`.

Procedure tinGame

Listing 4.14: Design specification of procedure tinGame

```

1  /**
2   * Performs coffee tin game to determine the last bean.
3   *
4   * @requires tin neq null /\ tin.length > 0
5   * @modifies tin
6   * @effects <pre>
7   *   repeat take out two beans from tin
8   *     if same colour
9   *       throw both away, put one blue bean back
10  *     else
11  *       put green bean back
12  *   until tin has less than 2 beans left
13  *   let p0 = initial number of green beans
14  *   if p0 = 1

```

```

15 *     result = 'G'
16 * else
17 *     result = 'B'
18 * </pre>
19 */
20 private static char tinGame(char[] tin)

```

This procedure has a parameter `char [] tin`, which represents the coffee tin. Each bean in the tin is represented either by the character ‘B’ (blue) or the character ‘G’ (green). The return type of the procedure is thus also a `char`, which represents colour of the last bean.

The `@requires` clause states that `tin` is initialised with at least one bean. In Java, a variable is `null` if it is an object-typed variable that has not been initialised. We will study objects in more detail later. It suffices to know that arrays are objects.

The procedure has a side-effect on `tin`, which is that it removes beans from the tin. The exact nature of the modification is described in `@effects`. This clause consists in two parts. The first part is basically high-level pseudocode in Listing 4.11, which describes the behaviour of the game. The second part, which starts with the `let` statement, makes precise the expected result of the procedure. This is determined based on the property of the last bean.

Procedure `hasAtLeastTwoBeans`

This procedure takes a `tin` and returns `true` or `false` depending on whether or not it has at least two beans. Listing 4.15 shows the design specification.

Listing 4.15: Design specification of procedure `hasAtLeastTwoBeans`

```

1 /**
2 * @effects
3 * if tin has at least two beans
4 *   return true
5 * else
6 *   return false
7 */
8 private static boolean hasAtLeastTwoBeans(char[] tin)

```

Procedure `takeTwo`

This procedure takes out two beans from `tin` and returns them in a `char` array. Listing 4.16 shows the design specification. The `@requires` clause states a condition

that `tin` must contain at least two beans. This assumption can be made in the context of `tinGame`, at the point after `hasAtLeastTwoBeans` has been performed. The `@effects` clause makes precise the behaviour by showing two invocations on `takeOne`, which pull out the two beans.

Listing 4.16: Design specification of procedure `takeTwo`

```

1 /**
2 * @requires tin has at least 2 beans left
3 * @modifies tin
4 * @effects
5 * remove any two beans from tin and return them
6 * i.e.
7 * char b1 = {@link takeOne(char[])} on tin
8 * char b2 = {@link takeOne(char[])} on tin
9 * result = [b1, b2]
10 */
11 private static char[] takeTwo(char[] tin)

```

Procedure `takeOne`

This procedure takes out any bean from `tin` and returns it. Listing 4.17 shows the design specification of the procedure. The `@requires` clause states a condition that `tin` has at least one bean left. This assumption can be made in the context of procedure `takeTwo`, which initially assumes that at least two beans are left in `tin`. As shown above, procedure `takeTwo` has two invocations on `takeOne`, which guarantee that the condition holds before each invocation.

Listing 4.17: Design specification of procedure `takeOne`

```

1 /**
2 * @requires tin has at least one bean
3 * @modifies tin
4 * @effects
5 * remove any bean from tin and return it
6 */
7 private static char takeOne(char[] tin)

```

Procedure `putIn`

This procedure puts back bean into `tin` at any vacant position. Listing 4.18 shows the design specification of the procedure. The `@requires` clause states a condition that

tin must have such vacant positions. This assumption can be made in the context of procedure `tinGame`, at the point after `takeTwo` has been invoked.

Listing 4.18: Design specification of procedure `putIn`

```

1 /**
2 * @requires tin has vacant positions for new beans
3 * @modifies tin
4 * @effects
5 *   place bean into any vacant position in tin
6 */
7 private static void putIn(char[] tin, char bean)

```

Procedure `anyBean`

This procedure looks in `tin` to find any bean left in it and return such bean. If no beans are found then it returns `NULL`, which is the Unicode's null character. Listing 4.19 shows the design specification of the procedure.

Listing 4.19: Design specification of procedure `anyBean`

```

1 /**
2 * @effects
3 *   if there are beans in tin
4 *     return any such bean
5 *   else
6 *     return NULL
7 */
8 private static char anyBean(char[] tin)

```

4.6.4 Code

Listing 4.20 shows the Java code of the program `CoffeeTinGame`.

Listing 4.20: The Java code of program `CoffeeTinGame`

```

1 package chap4.implement;
2 import java.util.Arrays;
3 import utils.TextIO;
4 /**
5 * @overview A program that performs the coffee tin game on a
6 *   tin of beans and display result on the standard output.
7 *
8 * @author dmle
9 */
10 public class CoffeeTinGame {

```

```

11  /** constant value for the green bean*/
12  private static final char GREEN = 'G';
13  /** constant value for the blue bean*/
14  private static final char BLUE = 'B';
15  /** constant for removed beans */
16  private static final char REMOVED = '-';
17  /** the null character*/
18  private static final char NULL = '\u0000';
19
20 /**
21 * the main procedure
22 * @effects
23 *   initialise a coffee tin
24 *   {@link TextIO#printf(String, Object...)}: print the tin content
25 *   {@link @tinGame(char[])}: perform the coffee tin game on tin
26 *   {@link TextIO#printf(String, Object...)}: print the tin content again
27 *   if last bean is correct
28 *     {@link TextIO#printf(String, Object...)}: print its colour
29 *   else
30 *     {@link TextIO#printf(String, Object...)}: print an error message
31 */
32 public static void main(String[] args) {
33     // initialise some beans
34     char[] tin =
35         { BLUE, BLUE, BLUE, GREEN, GREEN, GREEN };
36
37     // count number of greens
38     int greens = 0;
39     for (char bean : tin) {
40         if (bean == GREEN)
41             greens++;
42     }
43
44     final char last = (greens % 2 == 1) ? GREEN : BLUE;
45     // p0 = green parity \/
46     // (p0=1 -> last=GREEN) /\ (p0=0 -> last=BLUE)
47
48     // print the content of tin before the game
49     TextIO.printf("tin before: %s %n", Arrays.toString(tin));
50
51     // perform the game
52     char lastBean = tinGame(tin);
53     // lastBean = last \/ lastBean != last

```

```

54
55     // print the content of tin and last bean
56     TextIO.putf("tin after: %s %n", Arrays.toString(tin));
57
58     // check if last bean as expected and print
59     if (lastBean == last) {
60         TextIO.putf("last bean: %c ", lastBean);
61     } else {
62         TextIO.putf("Oops, wrong last bean: %c (expected: %c)%n", lastBean, last
63             ;
64     }
65
66 /**
67 * Performs the coffee tin game to determine the colour of the last bean
68 *
69 * @requires tin is not null /\ tin.length > 0
70 * @modifies tin
71 * @effects <pre>
72 *   take out two beans from tin
73 *   if same colour
74 *     throw both away, put one blue bean back
75 *   else
76 *     put green bean back
77 *   let p0 = initial number of green beans
78 *   if p0 = 1
79 *     result = 'G'
80 *   else
81 *     result = 'B'
82 *   </pre>
83 */
84 private static char tinGame(char[] tin) {
85     while (hasAtLeastTwoBeans(tin)) {
86         // take two beans from tin
87         char[] takeTwo = takeTwo(tin);
88         char b1 = takeTwo[0];
89         char b2 = takeTwo[1];
90         // process beans to update tin
91         if (b1 == BLUE && b2 == BLUE) {
92             // put B in bin
93             putIn(tin, BLUE);
94         } else if (b1 == GREEN && b2 == GREEN) {
95             // put B in bin

```

```

96         putIn(tin, BLUE);
97     } else { // BG, GB
98         // put G in bin
99         putIn(tin, GREEN);
100    }
101 }
102 return anyBean(tin);
103 }
104
105 /**
106 * @effects
107 * if tin has at least two beans
108 *   return true
109 * else
110 *   return false
111 */
112 private static boolean hasAtLeastTwoBeans(char[] tin) {
113     int count = 0;
114     for (char bean : tin) {
115         if (bean != REMOVED) {
116             count++;
117         }
118         if (count >= 2) // enough beans
119             return true;
120     }
121     // not enough beans
122     return false;
123 }
124
125 /**
126 * @requires tin has at least 2 beans left
127 * @modifies tin
128 * @effects
129 * remove any two beans from tin and return them
130 */
131 private static char[] takeTwo(char[] tin) {
132     char first = takeOne(tin);
133     char second = takeOne(tin);
134     return new char[] {first, second};
135 }
136
137 /**
138 * @requires tin has at least one bean

```

```

139  * @modifies tin
140  * @effects
141  *   remove any bean from tin and return it
142  */
143  private static char takeOne(char[] tin) {
144      for (int i = 0; i < tin.length; i++) {
145          char bean = tin[i];
146          if (bean != REMOVED) { // found one
147              tin[i] = REMOVED;
148              return bean;
149          }
150      }
151      // no beans left
152      return NULL;
153  }
154
155 /**
156 * @requires tin has vacant positions for new beans
157 * @modifies tin
158 * @effects
159 *   place bean into any vacant position in tin
160 */
161  private static void putIn(char[] tin, char bean) {
162      for (int i = 0; i < tin.length; i++) {
163          if (tin[i] == REMOVED) { // vacant position
164              tin[i] = bean;
165              break;
166          }
167      }
168  }
169
170 /**
171 * @effects
172 *   if there are beans in tin
173 *   return any such bean
174 * else
175 *   return '\u0000' (null character)
176 */
177  private static char anyBean(char[] tin) {
178      for (char bean : tin) {
179          if (bean != REMOVED) {
180              return bean;
181          }

```

```

182     }
183     // no beans left
184     return NULL;
185 }
186 }
```

4.6.4.1 Program constants

Since the implementation frequently needs to refer to two types of beans, we define them as the constants GREEN and BLUE in the program. In addition, constant REMOVED is used to mark a vacant position in the tin. Constant NULL represents the null character.

4.6.4.2 Procedure main

First, a `tin` variable is initialised with some beans. It is recommended to use different bean arrays here to test the program. Second, the expected last bean (variable `last`) is computed from `tin`, so that it can be used to check the game result later. This computation is based on the parity of the number of green beans as explained earlier. The next part of the code trivially realises the remaining tasks of the behaviour. Note that to display an array, the code first invokes procedure `Arrays.toString` to convert it into a string literal. This procedure is provided by class `java.util.Arrays`.

4.6.4.3 Procedure tinGame

The code consists in a `while` loop which realises the repetition of the game moves. The loop condition invokes the `hasAtLeastTwoBeans`. Each loop iteration represents a game move. First, it takes out two beans from `tin` by invoking `takeTwo`. Second, it invokes `putIn` with either ‘B’ or ‘G’ depending on the rule on the bean colors.

After the loop terminates, `anyBean` is invoked on `tin` to get any bean left in `tin`. Note that this procedure may return `NULL`, possibly because the above game moves were incorrectly performed.



Chapter 4 Exercise

These exercises are designed to help practise developing a number of small verifiable programs. You must use the development method explained in this chapter. You will find that although most of these programs are familiar and supposedly simple, constructing a good behaviour specification for them is not trivial!

1. Write the design specification for a Java program called `Arrays` that contains procedures for solving the following problems (one procedure per problem). The procedures must be named as given. The behaviour of procedure `main` of this program must be to display a list of the problems and prompt the user to choose which problem to solve. After the user has selected a problem, the procedure must prompt the user to enter input data for the selected problem. Finally, it invokes the procedure defined for the selected problem, passing in the input data, and displays the result on the standard output.
 - (a). `countNegatives`: count the number of elements of an array of integers that are negative
 - (b). `countEvens`: count the number of even elements of an array of positive integers
 - (c). `divArray`: divide the elements of a real number array by a real number
 - (d). `min`: find the minimum element in an array of integers
 - (e). `isAscSorted`: determine whether an array of integers is in ascending order
 - (f). `length`: find the length of an array of CHARs on the understanding that if it contains the character NUL (the character ‘\u0000’ in Java), assumed predefined as a constant, then that and any characters after it are not to be counted. In other words, NUL is understood as a terminator
 - (g). (*) `median`: find the median of an array of reals, that is the array value closest to the middle in the sense that as many array elements are smaller than it as are greater than it. Is the problem any easier if the array is known to be sorted?
 - (h). `compare`: given two arbitrary arrays of reals, a and b , determine if $a \subset b$, $a \supset b$, $a \cap b$, or $a = b$
 - (i). `freq`: compute the frequencies of each element of an array of reals
2. Implement the program `Arrays` that you designed in [Exercise 4.1](#). Run the program with some test data to make sure that it works correctly.
3. Write the design specification for a Java program called `Math` that contains procedures for solving the following problems. The procedures must be named as given. The behaviour of the procedure `main` of this program is similar to that of

program `Arrays` above.

- (a). `remainder`: determine the remainder after integer division using only subtraction. Ignore the possibility of division by zero.
 - (b). `div`: determine the integer division using only addition and subtraction. Ignore division by zero.
 - (c). `middle`: determine the middle one of three numbers
 - (d). `isPalindrome`: determine whether or not a string is a palindrom (a palindrom reads the same backward and forward, e.g. `deed`)
 - (e). `isPrime`: determine if an integer is a prime
4. Implement the program `Math` that you designed in [Exercise 4.3](#). Run the program with some test data to make sure that it works correctly.
 5. Extend of the programs `Arrays` and `Math` so that they run continuously, allowing the user to perform any number of computations, until (s)he enters a special input (e.g. character “Q” or “X”) to stop.

Chapter 5

Introduction to Object Oriented Program

Objectives

- ✓ Understand the concept of object oriented program (OOP) and its benefits.
- ✓ Compare and contrast between OOP and procedural program.
- ✓ Explain the core concepts of OOP: object, class.
- ✓ Describe the object oriented program development process.

This chapter marks the beginning of our treatment of a major topic of this book – object oriented program development. We shift our focus to the next level of abstraction in program design: from procedural abstraction to *data abstraction* [20]. We will study “object-oriented” data types that describe the information and behaviour of objects.

Conceptually, it is argued that designing a program in terms of objects is easier since it fits more naturally with the designer’s perception of the problem domain. For example, instead of considering a greeting conversation between two persons as a procedure that displays two greeting messages (one from each), it is more natural to think of it as a conversational situation in which two persons (or speaking more technically two person objects) greet one another with a message. What is interesting from this view point is that everything in a program is an object, including the conversation itself.

There are important benefits that can be gained from developing object oriented programs. But doing so requires support from the programming language in terms of how to represent objects, and, more generally, class of objects.

The purpose of this chapter is to provide a gentle introduction to object oriented program and its development. First, we define the basic concepts, including object oriented program, object and class. Next, we compare between object oriented program and the procedural counterpart. After that, we give an overview of an object oriented program development method.

5.1 Motivating Example

To understand the motivation for the development of OOP, let us consider a simple greeting conversion program. The objective of the program is to display greeting messages from a person group, whose names are specified from the standard in-

put. The greeting message takes the simple form “Hello, my name is X ”, where X is name of the speaking person. Listing 5.1 shows a procedural program named GreetingConversationProc1 that implements these requirements. We add a number prefix to this program because several versions of it will be developed in this chapter.

Listing 5.1: Procedural program: GreetingConversationProc1

```

1 import utils.TextIO;
2 /**
3  * @overview A program that displays greetings of a person group.
4 */
5 public class GreetingConversationProc1 {
6 /**
7  * @effects a group of persons whose names in args
8  * greet each other;
9  * i.e. for each name n in args
10 *      invoke greet(n)
11 */
12 public static void main(String[] args) {
13     if (args == null || args.length < 2) {
14         System.out.println("Requires at least 2 names");
15         System.exit(1);
16     }
17     for (String name: args) {
18         greet(name);
19     }
20 }
21 /**
22  * @effects displays "Hello, my name is " + name
23 */
24 private static void greet(String name) {
25     System.out.printf("Hello, my name is %s%n", name);
26 }
27 }
28 }
```

5.2 Why Object Oriented Program?

We will explain the motivation why we need **object oriented program (OOP)** by explaining the limitations of **procedural program (PPR)** and how OOP helps overcome those. To illustrate, we will use program GreetingConversationProc1 in Listing 5.1 as the basis and incrementally add some additional requirements to it. Since the be-

haviour is relatively straight-forward to implement (mostly involving printing a suitable message to the output), we will work primarily with the design specification and omit the program code. We will highlight the changes that are made to the design with **this font**.

5.2.1 Example: Greeting Conversation 2

The first requirement that we will add is the following: (*R1*) ***In addition to greeting, the person group also shake hands.*** This leads us to the definition of another procedure name `shakeHand` and update the procedure `main`'s behaviour to invoke this procedure. We name the new program `GreetingConversionProc2` and shows its specification in Listing 5.2.

Listing 5.2: Program `GreetingConversionProc2`

```

1 /**
2 * a group of persons greet and shake hand
3 * @effects
4 *   for each n in args
5 *     greet(n)
6 *   for each m in args, m neq n
7 *     shakeHand(n,m)
8 */
9 public static void main(String[] args)
10
11 /** @effects display a greeting message */
12 public static void greet(String name)
13
14 /** @effects display persons that shake hand */
15 public static void shakeHand(String name1, String name2)
```

5.2.2 Limitations of Procedural Program

Booch [4] suggests that PPR has the following three key limitations:

Problem 1. Does not effectively **model** the problem domain.

Problem 2. Not responsive to **change**.

Problem 3. Inadequate for problem domains with **concurrency**.

Let us illustrate these limitations with the greeting conversation example. Regarding **Problem 1**, persons are represented only by their names and so we can not differentiate two persons if their names are the same. Regarding **Problem 2**, it is difficult to update the program when changes occur. For example, both `greet` and `shakeHand` are affected if

name's data type is changed from String to char []. The impact is significantly larger in programs that have many methods referencing the same data type.

Regarding to [Problem 3](#), supporting concurrency requires splitting the program code among threads of execution. This is a serious issue if threads are run on different physical processors. To illustrate this, let us add another requirement to the original GreetingConversationProc1 in Listing 5.1: **(R2) In addition to greeting, the person group record names of each other into their mobile phones.** Listing 5.3 shows the specification of the updated program GreetingConversationProc3. Since name recording action can be performed at the same time as the person saying the greeting message, we would enforce that both actions are performed on a separate thread of execution. To achieve this, however, requires physically moving the two procedures greet and recordName to different threads.

Listing 5.3: Program GreetingConversationProc3

```

1 /**
2  * a group of persons greet each other and
3  * concurrently record names into phones
4  * @effects
5  *   for each name n & phone h of n in args
6  *     invoke greet(n) on th1,
7  *     recordName(h,m) on th2 for some name m neq n
8 */
9 public static void main(String[] args)
10
11 /** @effects ...as before... */
12 public static void greet(String name)
13
14 /** @effects record (name, phone) pair */
15 public static void recordName(String phone, String name)

```

5.2.3 How Does Object Oriented Program Help?

Figures 5.1 and 5.2 visually explain how the three problems of PPR can be overcome in OOP. The LHS of Figure 5.1 shows the design specification of an equivalent OOP for the basic requirement of GreetingConversationProc1. It shows how the concept of Person is captured directly by a data type named Person. The ability to represent real-world domain concepts directly in the design helps ease problem solving. This diagram also shows that instead of accessing the person name directly, the behaviour greet of Person is invoked. Hiding person name behind this behaviour eases maintenance

because changing name's data type does not affect the client code.

```
/*
 * a group of persons greet
 * @effects
 *   for each Person p
 *     invoke p.greet()
 */
public static void main(String[] args)
  P2: name is hidden behind
      behaviour invocation

  P1: models
      real-world
      concept
      PERSON

  P2: greet and
      shakeHand are not
      affected if name's
      data type is changed
*/
public static void main(String[] args)
```

Figure 5.1: Overcoming Problem 1 and Problem 2 with OOP.

```
/*
 * a group of persons greet and shake hand
 * at the same time.
 *
 * @effects
 *   for each Person p, Phone h of p
 *   invoke p.greet() on th1
 *   invoke h.recordName(q) on th2 for some
 *   Person q neq p
public static void main(String[] args)
```

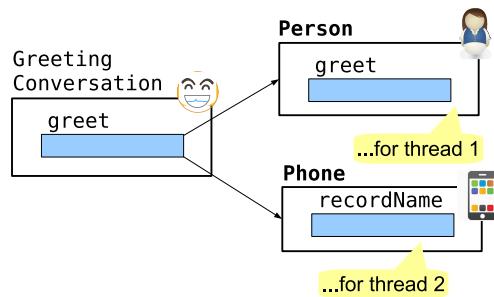


Figure 5.2: Overcoming Problem 3 with OOP.

Data hiding is further demonstrated in the RHS of Figure 5.1, which shows the design specification of an OOP version of GreetingConversationProc2. In this figure the two actions (greet and hand-shaking) are performed via behaviour invocations, without any direct references to name.

The LHS of Figure 5.2 shows the design specification of an OOP version of GreetingConversationProc3. It illustrates the ease of support for concurrent design. The two actions greet and recordName are defined on two different object types (Person and Phone, *resp.*). Thus, invoking these two behaviours on two threads (th1 and th2) simply amounts to mapping the two object types to the threads. Further, if the two threads are physically located on two separate devices (such as the situation shown in the RHS of Figure 5.2) then the object codes can also be written to these devices accordingly. The object type makes the mapping to real-world objects easy.

5.2.4 Historical Context

According to Booch [4], object oriented program development is the result of or influenced by the following developments:

- object-oriented computer architectures: e.g. SWARD, Intel 432 and IBM System 38 are object-oriented architectures.

- advances in programming languages to express higher-level abstractions (e.g. Simula, Pascal, Smalltalk, CLU, and Ada) and to approach closer to realising the object-oriented concepts:
 - Simula 67 introduced class for data encapsulation
 - SmallTalk defined the term “object-oriented”
- abstract data type (ADT) research works, especially from the authoritative figures (Liskov, Guttag, and Shaw)

A parallel but related development to object-oriented development is ADT. Booch [4] suggests that OOP was developed on the back-drop of ADT. ADT is similar to class (object type) in that it also defines the behaviour of objects. However, ADT is concerned primarily with passive objects, i.e. those that only perform actions when requested by others. ADT does not address the interaction between objects. An ADT only specifies the behaviour that objects have. It does not define the behaviour that an object requires others to possess.

5.3 What Is OOP?

An OOP operates at a level of abstraction higher than that of PPR. The following definition of OOP, which is adapted from Booch [4], makes clear the core abstractions of the program.

Definition 5.1. name

*An **object oriented program (OOP)** is a set of objects interacting with each other to achieve the program’s intended behaviour.*



Booch [4] defines **abstraction** as “... a simplified description, or specification of a system that emphasises some of the system’s details or properties while suppressing others”. This definition is consistent in meaning with that used in an earlier chapter to define procedural abstraction. Unlike PPR, however, the abstractions of an OOP are *objects* and *classes*. We will give below the intuitive descriptions of these two terms. More technical descriptions of these concepts will be given later in Section 5.4.

An **object** describes a particular thing of interest that our program seeks to capture as input, process, or generate as output. For example, there are two person objects that the greeting conversation program seeks to capture as input. Objects are not isolated components of a program but **interact** with each other through behaviour invocation to achieve the program’s intended behaviour.

A **class** is a set of all objects in the problem domain that share similar features. For

example, in the greeting conversation problem the person objects share similar behaviour and thus can be represented by a class named Person.

Example 5.1 Object oriented program: GreetingConversation.

Listings 5.4 and 5.5 show the code of two classes that make up an OOP version of the greeting conversation program. In Listing 5.4, class GreetingConversation represents greeting conversations between persons. This class has two procedures: main, and greet. Procedure main is the executable procedure of the class. Procedure greet, however, is not a stand-alone procedure. This procedure creates two Person objects, using the class Person defined in Listing 5.5, and invokes procedure Person.greet upon these objects to display the greeting messages specific to each.

Listing 5.4: Class: GreetingConversation

```

1 package chap5;
2 /**
3 * @overview A program that displays greetings of a person group.
4 * @author dmle
5 */
6 public class GreetingConversation {
7     /**
8     * @effects
9     * Create a greeting conversation between a person group
10    * whose names are specified in <tt>args</tt>
11   */
12   public static void main(String[] args) {
13       GreetingConversation gc = new GreetingConversation();
14       gc.greet(args);
15   }
16
17   /**
18   * @effects
19   * Create a person group from specified <tt>names</tt>
20   * and make them greet each other
21   */
22   public void greet(String[] names) {
23       if (names == null || names.length < 2) {
24           System.err.println("Requires at least 2 names");
25           return;
26       }
27
28       // create person group
29       Person[] persons = new Person[names.length];
30       for (int i = 0; i < names.length; i++) {

```

```

31     String n = names[i];
32     Person p = new Person(n);
33     persons[i] = p;
34 }
35
36 // make them greet
37 for (Person p : persons) p.greet();
38 }
39 } // end GreetingConversation

```

Listing 5.5: Class: Person

```

1 package chap5;
2 /**
3 * @overview Represents persons with a name attribute.
4 * @author dmle
5 */
6 public class Person {
7     private String name;
8
9     /**
10      * @effects initialise this as an object whose name is aName
11      */
12     public Person(String aName) {
13         name = aName;
14     }
15
16     /**
17      * @effects updates name with newName
18      */
19     public void setName(String newName) {
20         name = newName;
21     }
22
23     /**
24      * @effects Prints a greeting message containing name to
25      *          the standard console
26      */
27     public void greet() {
28         System.out.printf("Hello, my name is %s%n", name);
29     }
30 }

```



5.4 Object and Class

First, we adopted the following definition about object from Booch et al. [5].

Definition 5.2. Object

*An **object** describes an entity (a particular thing of interest) that has state, behaviour and identity. The structure and behavior of similar objects are defined in their common class. The terms **instance** and **object** are interchangeable.*



To construct objects in an OOP the concept of class is introduced. Class represents the design blueprint for objects. An object is an instance of a class, created to capture a particular thing of interest in the problem domain.

Definition 5.3. Class

*A **class** is a set of similar but unique objects in a problem domain [4]. Class defines the data attributes and behaviour of its objects.*



For example, in the program GreetingConversation, person objects are constructed using a class named Person¹. This class was shown earlier in Listing 5.5. In general, an OOP would be constructed from several classes, whose objects interact with each other to achieve the program's behaviour. For example, greeting conversation program has another class named GreetingConversion, shown in Listing 5.4.

To understand objects requires to think more abstractly about data. Objects with data and behaviours are a higher-level of abstraction than procedures. First, objects have characteristics, a.k.a **attributes**, and that data are specific values that are assigned to these attributes. These (attribute, value) pairs form the object **state** [5]. Objects of the same class have their own states, independently from each other. For example, two person objects in program GreetingConversation have different values for the attribute name. Further, object state (more precisely the attribute values) typically changes over time, depending on the operations that it needs to perform.

More than state, however, objects also have behaviours. A **behaviour** is an abstraction of an action or task that an object performs, which typically requires using the object state in some way. A more technical definition of object behaviour is given by Booch et al. [5] as follows: “behavior is how an object acts and reacts, in terms of its state changes and message passing”. Message passing is synonymous to behaviour invocation. From this view point, object behaviour is the sum of all the *object procedures* [5], i.e. procedures that must be invoked upon an object. For example, in

¹As a convention, the first letter of every word of a class name is capitalised.

program GreetingConversation, class Person defines an object procedure named greet, which generates a greeting message based on the value of the attribute name.

Objects are independent but not isolated components of a program. They **interact** with each other through behaviour invocation to achieve the program's intended behaviour.

Definition 5.4. Object interaction

Object interaction consists in an object invoking the behaviours of some other object(s) that may be of the same class.



Example 5.2 Objects and classes in GreetingConversion.

As shown in Example 5.1, GreetingConversation objects require the behaviour Person.greet – the greet behaviour of Person objects – to function correctly. When invoked upon a Person object, this procedure displays a greeting message using value of the object's name. Procedure GreetingConversation.main invokes procedure GreetingConversation.greet via a GreetingConversation object. This procedure in turn performs several invocations upon procedure Person.greet via objects of the Person class.

Person objects have restricted visibility in the sense that the values of their name attribute are hidden and cannot be accessed directly by the program. This is achieved in Java using the keyword `private` in the definition of the attribute. The only way to change a Person object's name is to use the object procedure `setName`.

Similar to the procedural program counterpart, GreetingConversation is defined such that the behaviours of its procedures are defined in the program specification, separated from the implementation. This specification forms the outside view, as they allow GreetingConversation objects to access and manipulate the data. The implementations of these procedures, however, form the internal view. □

Object Procedure vs. Class Procedure

As discussed above, object procedure is a procedure that must be invoked upon an object of the class. On the other hand, class procedure (a.k.a stand-alone procedure in Chapter 4) is a procedure that is invoked upon the class and, as such, does not require any objects of the class to be created. In Java, an object procedure is defined without the keyword `static`, while a class procedure is defined with this keyword.

For example, class GreetingConversation in Listing 5.4 has one class and one object procedure. The class procedure is `main` as it is invoked by the Java run time, without requiring an object of this class to be created. The object procedure is `greet`,

which is invoked by the procedure `main` via a `GreetingConversation` object.

Class `Person` has two object procedures: `greet` and `setName`, both of which cannot be changed to class procedures. The reason is because these procedures operate over the value of attribute `name`, which is specific to each object of the class.

5.5 Class Design Diagram

It is useful to use a diagrammatic notation to represent class. Figure 5.3 shows one commonly-used notation that is used by an object oriented design language called **Unified Modelling Language (UML)** [22]. The class depicted in the figure is the class `Person` of the greeting conversation program.



Figure 5.3: The design diagram of class `Person`.

As shown in the figure, a class is represented by a labelled rectangle consisting of three parts. The top part contains the name of the class (which is “`Person`” in this case). The middle part lists the attribute names and their data types. Each entry has the form `attribute_name: data_type`. The minus sign ‘-’ before each entry means that the attribute is private. For example, the diagram shows that class `Person` has a private attribute called `name`, whose data type is `String`.

The bottom part lists the object procedures, which are called **operations** in UML. An operation has the format: `procedure_name(parameter_list): return_type`. The part “`: return_type`” can be omitted if a procedure it does not return any values. The plus sign ‘+’ before an entry means that the operation is public.

For instance, the figure shows that class `Person` has two public operations²: `setName` and `greet`. The former takes a `String` object as input, while the latter does not take any input. Both operations do not return any values.

²another operation `Person(String)` is omitted from the figure and will be discussed later.

5.6 Information Hiding

Information hiding means to hide the internal implementation detail of an object, so that internal design changes do not affect other objects. In the greeting conversation program, for example, it is not necessary for `GreetingConversation` object to know how the operation `Person.greet` is implemented. All it needs to do is what this operation does (its behaviour) in order to use it correctly.

For example, information hiding is achieved in the class `Person` by hiding the definition of attribute `name` (structural elements) and implementations of the public operations. The former is achieved through the access modifier `private`. In contrast to `public` members, a `private` member of a class is only accessible to procedures of the same class³. The only way to change the value of attribute `Person.name` is by invoking operation `setName`.

5.7 Comparing OOP with Procedural Program

If this chapter is to serve as a transition from PPR to OOP then this and the next section help clarify the purpose. So far in this chapter, we have highlighted some similarities between OOP and PPR. In particular, both types of program consist of classes and classes consist of procedures. Further, the overall program's intended behaviour is achieved generally through procedure invocation.

However, a key difference between the two program types concerns the notion of object. The classes of a PPR are simply containers for the functional procedures, while those of an OOP represent data types that are used to create objects. In the greeting conversation program, for instance, the PPR version consists of one class, `GreetingConversationProc`, which is used as container for the two stand-alone procedures `main` and `greet`. These procedures are called directly, without needing any objects of the class. In the OOP program `GreetingConversation`, however, both classes `GreetingConversation` and `Person` are used as data types to create objects. The ability to create new data types that capture the domain requirement is a key benefit of OOP compared to PPR.

³to be more precise, since `Person.name` is non-static, only non-static procedures of class `Person` can access it.

5.8 Benefits of OOP

There are several important benefits that can be gained from developing programs the object-oriented way. From the analysis and design's point of view, object oriented program development eases **problem solving**. Since objects exist naturally in the problem domain, structuring a problem in terms objects makes it easier to understand and to solve. Further, OOPs can enhance program **modularity** through information hiding. Modularity increases **reuse** and eases program **maintenance**. The same objects can be reused in other programs that require their behaviour. Modification of a program element can be localised to the internal parts without affecting other parts.

Another benefit of OOP is that it provides better support for problems that require concurrency [4]. This capability was demonstrated in Section 5.2.

5.9 OOP Development Process

In this section, we will give an overview of the development process of an OOP. Our objective at this stage is not to understand the process in detail but to appreciate overall picture and how the basic concepts that we discussed in this chapter are applied in practice. This will provide a context for the subsequent chapters, where we discuss in detail class design and implementation. In more advanced topics about OOP development (e.g. [5, 17, 30]), the development process actually consists of several phases, each phase in turn consists in several activities. For our study purpose here, we will focus on the later stages of design and implementation. The materials for this section are mostly drawn from [4, 5]. However, the development process will be presented using more modern object oriented terms and techniques. In particular, we will use the UML language [22] to draw the software models.

5.9.1 Example: Floating buoys management software

To illustrate the OOP development process, this section uses a more complex, real-world example. Below is a description of the program requirements, taken directly from [4]:

There exists a collection of free-floating buoys that provide navigation and weather data to air and ship traffic at sea. The buoys collect air and water temperature, wind speed, and location data through a variety of sensors. Each buoy may have a different number of wind and temperature sensors

and may be modified to support other types of sensors in the future. Each buoy is also equipped with a radio transmitter (to broadcast weather signal and location information as well as an SOS message), and a radio receiver equipped with a red light, which may be activated by a passing vessel during sea-search operations. If a sailor is able to reach the buoy, he or she may flip a switch on the side of the buoy to initiate an SOS broadcast. Software for each buoy must:

- maintain current wind, temperature, and location information; wind speed readings are taken every 30 seconds, temperature readings every 10 secs and location every 10 secs; wind and temperature values are kept at running average.
- broadcast current wind, temperature, and location information every 60 secs
- broadcast wind, temperature, and location information from the past 24 hours in response to requests from passing vessels; this takes priority over the periodic broadcast
- activate or deactivate the red light based upon a request from a passing vessel
- continuously broadcast an SOS signal after a sailor engages the emergency switch; this signal takes priority over all other broadcasts, and continues until reset by a passing vessel

5.9.2 Overview of the Development Method

The development method consists in the following steps that are performed in some order. The subsequent sections will explain these steps in more detail.

1. Create a function design model of the program
2. Identify the objects and attributes
3. Identify self and required behaviour
4. Define classes
5. Identify associations between classes
6. Write the design specification of each class
7. Implement each class

Except for steps 1 and 4, other steps are similar to those mentioned in [4]. Step 1 aims to structure the program requirements and to build a functional design model that describes the logic of the tasks that the program are to perform. This model, which

is constructed in such modelling languages as UML, also provides a visual aid for the design steps that follow. Steps 2 and 3 aim to identify and shape the objects that the program needs to capture and process. In step 4, suitable classes are then constructed for these objects. In step 5, the classes are updated with associations, which describe relationships between objects. The associations are identified using the functional design model constructed in step 1. In naming step 5, we changed the term “visibility” used in step 3 of [4] to “association”. This is the current terminology used in object oriented design [22] that has the same meaning as this phrase in [4] “...the static dependencies between classes”. Step 6 is to create the design specification for each class. The result of this step is used as input in the final step, which implements each class in the target OOPL.

Note that a thorough discussion of steps 1 to 5 is beyond the scope of this chapter. Such discussion can be found in more specialised text books either on object oriented software development (e.g. [5, 17]) or on software engineering (e.g. [30]). Steps 6 and 7 will be discussed in detail in the subsequent chapters of this book.

5.9.3 Create a function design model

We changed the design model language used in the initial step of [4] from data flow diagram to UML, and chose UML activity diagram to be the language for the function design model. The aim is to build a model that captures the activities (i.e. tasks) that the program is to perform, together with the exchanges of data and flows of control between them. These elements are needed to determine the objects and their behaviours.

The data flow diagram in [4] is transformed into an activity diagram with some modifications. For example, clock is modelled as passive rather than active. To assist the subsequent steps, we use the swimlane feature of activity diagram to organise the activities based on the actors that perform them.

Functional model as an activity diagram

The functional model in Figure 5.4 was prepared using an object oriented software design tool, named Visual Paradigm for UML⁴. The following points should be noted about this activity diagram:

- to conserve space, one Sensor actor is used to represent the four types of sensors
- the action names are chosen to be generic names, e.g. “calculate value”

⁴<http://www.visual-paradigm.com>

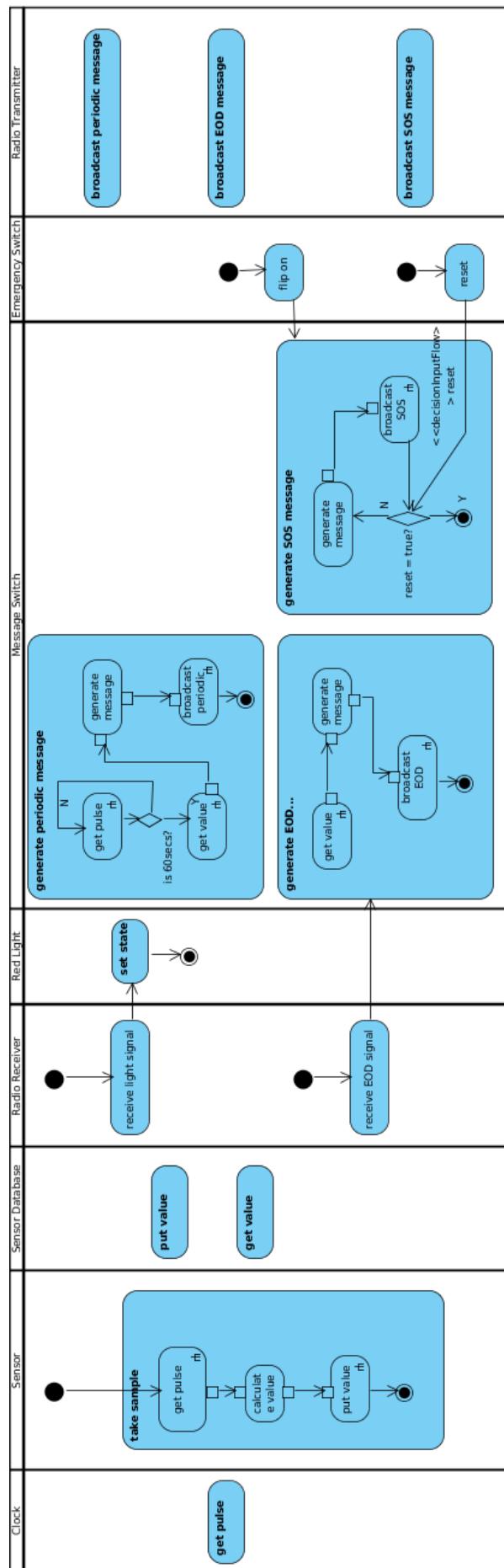


Figure 5.4: The activity diagram of the floating buoys management software.

- one vertical swimlane is used to group the activities performed by each actor
- 5 initial nodes, each is used to initiate a flow
- an activity final node is used to end each activity
- four activities are expanded with the details of the nodes and edges; four others (get pulse, set value, get value, set state) are not expanded
- call behaviour action (similar to procedural invocation but for behaviour in general) is annotated with a rake-style symbol:
 - each action invokes an activity with the same name (e.g. get pulse, get value, broadcast periodic message, . . .)
- one activity may be invoked by more than one actions (e.g. get pulse, get value)

5.9.4 Identify objects and attributes

In this step, the facts given in the program requirements about data are represented in terms of objects. We extended the guidelines in [4] to identify the objects and their attributes from the design model as follow:

- for each object flow, classify its data according to the following categories:
 - (1) primitive type, (2) value of some attribute, (3) object
 - do nothing for data that belongs to category (1)
 - if the data belong to category (2) then create or update an object containing that attribute
 - if the data belong to category (3) then create a new object
- for each activity, identify the object suitable for performing that activity:
 - use the actors involved in the swimlanes

Object definitions

Table 5.1 describes the objects that are identified for the example. The identified objects reflect the distributed nature of the underlying system: objects are deployed onto different devices that interact with each other, possibly in a concurrent manner, via communications networks. Note the followings about the table:

- “hardware-type” objects: due to the nature of the system, all objects except the message switch are hardware-based, i.e. software entities that operate inside the corresponding hardware devices
- four types of sensor objects are identified, one for each type of data
- clock object is created based on the domain knowledge of this kind of system

Table 5.1: Object definitions

Objects	Descriptions
clock	provides the stimulus for periodic actions
wind speed sensors	maintains a running average of wind speed
air temperature sensors	maintains a running average of air temperature
water temperature sensors	maintains a running average of water temperature
location sensor	maintains the current buoy location
sensor database	serves to store weather and location history
radio receiver	provides a channel for requests from passing vessels
radio transmitter	provides a channel for broadcast of weather and location reports as well as SOS messages
emergency switch	provides the stimulus for the SOS signal
red light	controls the activity of the emergency light
message switch	serves to generate and arbitrate various broadcast messages

Table 5.2: Self behaviour

Objects	Self behaviour
clock	get pulse
wind speed sensors	take sample
air temperature sensors	take sample
water temperature sensors	take sample
location sensor	take sample
sensor database	put value, get value
radio receiver	receive EOD (end-of-day) signal, receive light signal
radio transmitter	broadcast periodic message, broadcast EOD message, broadcast SOS message
emergency switch	flip on, reset
red light	set state
message switch	generate periodic message, generate EOD message, generate SOS message

- sensor database is not the actual database, but a software entity that is used by the program to access the database

5.9.5 Identify self and required behaviour

Self behaviour

This includes the operations that each object makes available for objects to invoke. These operations are defined based on the activities performed by the objects identified in the previous step.

Table 5.3: Required behaviour

Objects	Required behaviour
clock	-
wind speed sensors	get pulse, put value
air temperature sensors	get pulse, put value
water temperature sensors	get pulse, put value
location sensor	get pulse, put value
sensor database	-
radio receiver	force activate, force EOD message generation
radio transmitter	-
emergency switch	force SOS message
red light	-
message switch	get pulse, get value, broadcast periodic message, broadcast EOD message, broadcast SOS message

Behaviour definitions. Table 5.2 below describes the self behaviour of the example. The first column lists the objects and the second column lists the raw names of the operations performed by the objects. Note that this list is not exactly the same as the list in [4]. The objects whose behaviours differ from [4] are highlighted in bold:

- clock is considered passive (active in [4])
- radio receiver has three operations (no operations in [4])
- emergency switch has two operations (no operations in [4])
- message switch actively generates periodic messages (passively in [4])

Required behaviour

This is the behaviour that an object requires other objects to have in order to handle the activity flows that originate from it. It includes operations or stimuli that cause the invocations of some behaviour. The stimuli are written in the form “force...”.

Behaviour definitions. Table 5.3 describes the required behaviours of the example.

Note the following differences between this table and the list in [4]:

- sensor objects and message switch requires procedure “get pulse” from clock
- clock has no required behaviour since it is passive
- radio receiver: “set light state” is changed to “force activate”

5.9.6 Define classes

Now that the objects together with their attributes and behaviours have been identified, we are ready to construct the classes for these objects. The following guidelines

Table 5.4: Class definitions

Classes	Descriptions	Behaviour
Clock	Provides the stimulus for periodic actions	getPulse
WindSpeedSensor	maintains a running average of wind speed	takeSample
AirTemperatureSensor	maintains a running average of air temperature	takeSample
WaterTemperatureSensor	maintains a running average of water temperature	takeSample
LocationSensor	maintains the current buoy location	takeSample
SensorDatabase	serves to store weather and location history	putValue, getValue
RadioReceiver	provides a channel for requests from passing vessels	receiveEODSignal, receiveLightSignal
RadioTransmitter	provides a channel for broadcast of weather and location reports as well as SOS messages	broadcastPeriodicMessage, broadcastEODmessage, broadcastSOSMessage
EmergencySwitch	provides the stimulus for the SOS signal	flipOn, reset
RedLight	controls the activity of the emergency light	setState
MessageSwitch	serves to generate and arbitrate various broadcast messages	generatePeriodicMessage, generateEODMessage, generateSOSMessage

are used:

- objects with similar attributes and behaviour are grouped into the same class
- class names are singular nouns and written without spaces

Class definitions

Table 5.4 describes the classes for the example. As can be seen from the table, the classes correspond directly with the object types in Table 5.1. Note that reuse can be enhanced by grouping the sensors into a hierarchy of sensors as described in [4]. However, to simplify the discussion we will not discuss this here.

5.9.7 Identify associations

Associations are relationships among the classes which follow the directions of the behaviour required of their objects. The associations also show the dependencies

Table 5.5: Associations

Class A	Required behaviours	Class B
Clock	-	-
WindSpeedSensor	getPulse, put value	Clock, SensorDatabase
AirTemperatureSensor	getPulse, put value	Clock, SensorDatabase
WaterTemperatureSensor	getPulse, put value	Clock, SensorDatabase
LocationSensor	getPulse, put value	Clock, SensorDatabase
SensorDatabase	-	-
RadioReceiver	force activate, force EOD message generation	RedLight, MessageSwitch
RadioTransmitter	-	-
EmergencySwitch	force SOS message	MessageSwitch
RedLight	-	-
MessageSwitch	getPulse, getValue, broadcast periodic message, broadcast EOD message, broadcast SOS message	Clock, SensorDatabase, RadioTransmitter

between the classes.

Association definitions

Table 5.5 lists the associations for the classes in the example. The first and third columns list the classes involved in the associations. The second column lists the behaviours based on which the associations are defined.

5.9.8 Class diagram

Figure 5.5 is the class diagram that models the classes and their associations. This is another type of model supported by UML. The diagram was drawn using the UML2 drawing component of the Eclipse IDE⁵. In the figure, classes are shown as labelled rectangles and associations are as arrowed lines that connect the rectangles. An arrow describes the direction of the dependency: the class at the source of the arrow depends on the class at the arrowed end.

⁵<https://www.eclipse.org/modeling/mdt/?project=uml2>

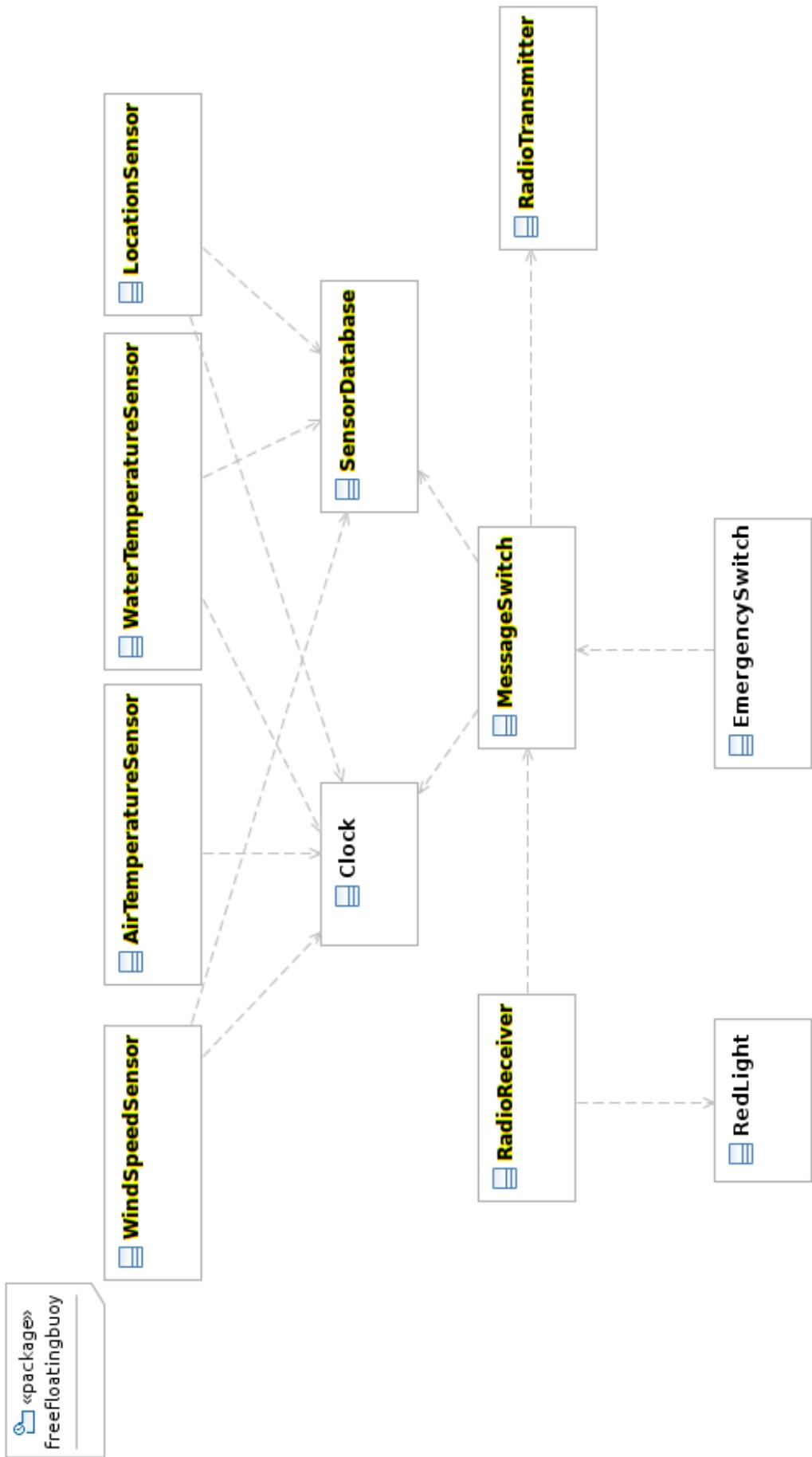


Figure 5.5: Class diagram of the buoy program.



Chapter 5 Exercise

The exercises in this chapter are designed to introduce the key concepts of OOP.

Note that the term “design” here refers only to the initial conceptual design of a program. The concrete design will be studied in later chapters. The main objective is to become familiar with the key OOP concepts, namely class and object. The key tasks include identifying class or classes that are needed to describe the concepts, drawing up the shape of each class using the UML class notation. This diagram need to include any attributes and operations that the class may have.

1. The greeting conversation program discussed in the lecture now needs to capture and process data about mobile phones. A mobile phone contains data about manufacturer name and the phone model. For example, the phone Samsung Galaxy 3S, GT-55360 has manufacturer name “Samsung” and the model “GT-55360”. A mobile phone can perform two actions: one is to change the model and the other is to record the name of a person.

Draw a design diagram for phone.

2. Extend the greeting conversion program further to capture data about which person owns which phone. For example, the program should be able to capture the fact that a person named “Duc” owns the Samsung phone mentioned in the previous task.

Think of at least two ways in which you can design this new requirement. Draw a suitable design diagram for each.

3. In this task, you will design an object oriented program for the procedural program Arrays that you wrote in a previous tutorial. In this OOP, which is called Array, an Array object has its state set to an array of integers. Each object can perform the following actions upon this array:

- (a). count the number of elements that are negative
- (b). find the minimum element
- (c). determine whether the array is in ascending order
- (d). return the length of the array
- (e). compute the frequencies of each element of the array

Draw a design diagram for Array.

4. What are the benefits of the OOP Arrays that you designed over the procedural version?
5. Write the code for the greeting conversation program that you designed in [Exercise 5.2](#).
6. Write the code for the program Array that you designed in [Exercise 5.3](#).

Chapter 6

Object Oriented Design Fundamentals

Objectives

- ✓ Understand the need for constructing a detailed design specification of class.
- ✓ Understand the difference and relationship between abstract and concrete design levels and how to define both in a specification. Apply the verifiable program design specification method for class.
- ✓ Specify a class to model a domain concept.
- ✓ Describe the essential domain constraint rules and the essential operation types. Explain how to choose suitable operations for a class.
- ✓ Apply an essential set of annotations to specify the design of attributes and operations.
- ✓ Explain the advantages and disadvantages of the alternative concrete data types of an attribute.
- ✓ Explain the difference between the design specifications of collection and non-collection classes.

As we saw in the previous chapter, object oriented program (OOP) relies on a strong understanding of the conceptual structure of the problem domain. A well-thought-out OOP design is thus necessary for a successful implementation in a target OOPL. Because of the particularly important role of OOP design, we devote three chapters of this book to it. In this chapter, we discuss class design – the fundamental building block of OOP. In two latter chapters, we will discuss a number of design issues and introduce some basic but useful OOP design patterns.

In this chapter, we first critically review a number of key concepts introduced in the literature that are about or related to class. After that we discuss how to apply the design specification method of procedural programs that we studied to design class. The core of the design specification language remains the same. However, we will introduce a number of essential extensions to this language in order to accommodate class and its members (namely fields and methods).

6.1 Motivation

As we have seen with procedural programs, design takes time and effort, which would affect the overall development timeline. The more detailed a design, the more time it would take to complete. So two natural questions to ask are *why do it?* and *how detailed does the design need to be?*

Let us consider the bare design of the class Person shown in Figure 6.1. We introduced this design in the previous chapter. Is this design sufficient for successful program implementation?

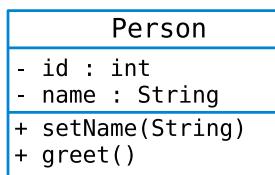


Figure 6.1: A bare design of class Person.

A closer look at this Person class design can reveal the following questions that still need to be answered before we can start coding it:

1. Can name be typed `StringBuilder`¹ instead of `String`?
2. Can `id` be negative?
3. Can `name` be uninitialised (i.e. takes `null`)?
4. How do we create a `Person` object with a given `id` and `name`?
5. How do we obtain the value of `name`?
6. Should there be an operation to change value of `id`?
7. Are there any other essential operations of this class?

The first three questions concern the design of the object state (attributes and their values) while the other four questions concern the essential operations that are needed in order for a client program to manipulate the objects and their states. These questions are important not only because they concern the essential design elements but because there are potentially alternative answers to each of them. Further, it is not always clear which answer is the best one to use in all situations. For instance, both data types mentioned in Question 1 are suitable for attribute `name`. The choice of which one to use depends on whether or not this attribute's value is often modified rather than replaced.

To make informed design decisions require capturing in the design model the necessary design rules from the problem domain. Unfortunately, the design model in Figure 6.1 does not capture these rules. In this chapter, we will study a design method

¹A mutable string data type provided by Java

for class that helps capture sufficient detail to answer the above and other important questions about the design.

Design Examples

To illustrate the concepts and techniques presented in this chapter, we will use two representative design examples. The first example describes a concept, named `Customer`, which is commonly found in business-typed software. At the bare minimum, a customer is characterised in terms of an identifier (*abbr.* `id`) and full name (*abbr.* `name`). The second example is an adaption of a collection-typed concept, named `IntSet`, which represents a set of integers. This concept is commonly used as a technical concept in a program to help organise data objects. `IntSet` is defined in Chapter 5 of the Liskov and Guttag's book [20].

6.2 Design Terms

In this section, we give a gentle overview of a number of basic class design concepts that are presented in [20]. We will attempt to relate the concepts and highlight their main underlying features. The LHS of Figure 6.2 presents a terminology map. The RHS of the figure illustrates with an example about a very popular real-world concept named customer (often found in business-typed programs). The illustration represents a realisation (or an instance) of the term map with regards to customer.

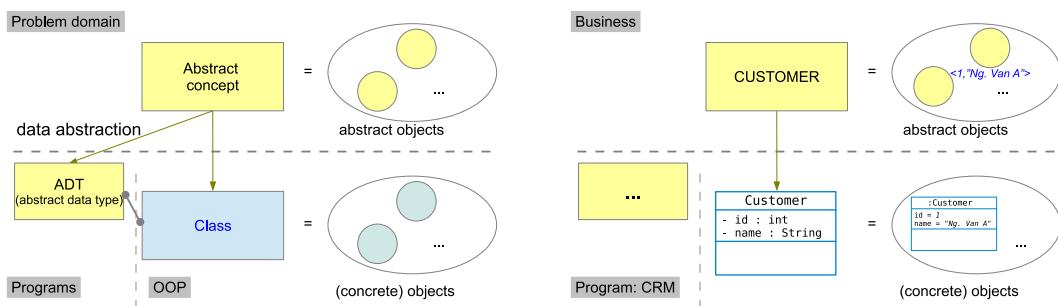


Figure 6.2: Design term map.

The rest of this section will discuss the following terms that are listed in the figure:

- data abstraction
- abstract concept
- class and abstract data type (ADT)
- object

6.2.1 Data abstraction

Data abstraction [20], or type abstraction in [13], is an abstraction that captures details about the data values and the operations that operate on them. It came out of the need to add new data types in programming. In this view, data abstraction differs from procedural abstraction because the latter results in no new types being added, only a functional extension to the way in which the built-in data types of a programming language are used.

In an early work [13], data abstraction was associated to abstract data type (ADT). Works by the same author [12, 20], however, associate the concept to the term class in object-oriented programming. Guttag [12] makes explicit this association by considering an ADT as “...a class of objects defined by a representation independent specification”. The meaning of *representation* will be explained later in this chapter.

6.2.2 Abstract concept

Abstract concept is a term used in [20] to refer to a data entity in the problem domain of a program. For example, POLYNOMIAL is an abstract concept in the algebraic problem domain, while CUSTOMER is an abstract concept in commercial problem domains. These concepts are what programs eventually realise in code. However, they are often created to model the problem domain before the programs are actually written. This design task is performed by the analyst/designer as part of a formal enquiry into the problem domain. The aim of this enquiry is to determine the suitable programming solutions for the domain.

An abstract concept also has instances, which are called **abstract object** [20]. For example, the customer whose `id` is 1 and whose `name` is “Nguyen Van A” is an abstract object of the abstract concept CUSTOMER.

6.2.3 Class

A **class**, in the object oriented programming terminology, is a particular model of an abstract concept that is constructed for an object oriented programming solution. In other words, when we want to develop an OOP to capture and process data about an abstract concept, we define a class to model that concept. Conversely, we say that the abstract concept provides meaning for the class.

For instance, we would create in an OOP a class named `Poly` [20] to model the algebraic concept POLYNOMIAL, and class `Customer` to model the concept CUSTOMER.

More formally, we adopt the following tuple-typed definition of class [20]:

```
class = <objects, operations>
```

That is, a class is a binary tuple defined over a set of objects and a set of operations that are performed on them.

Example 6.1 Class Let us consider two class examples taken from [20]. We will use these examples later in this note to demonstrate class design and implementation.

IntSet An integer set, denoted by IntSet, is a set of integer numbers. This class is the tuple: $\langle X, \text{OptsS} \rangle$, where X is the set of all integer sets and OptsS is a set of operations on X . OptsS contains the following operations on IntSet objects:

- add an element
- remove an element
- check membership
- determine the cardinality
- pick an arbitrary element

Customer Class Customer the tuple $\langle C, \text{OptsC} \rangle$, where C is the set of all customer objects and OptsC is a set of operations on C . OptsC contains the following operations on Customer objects:

- set id and name
- get id and name
- place a product order

□

6.2.4 Class vs. abstract data type

An **abstract data type (ADT)** is a data type that defines the data values manipulated by a program in terms of the operations that are performed on them [13]. Figure 6.2 shows that abstract concept can be realised in a program in terms of either an ADT or a class. ADT represents a set of realisation forms of the abstract concept in a program. Class is an object-oriented form of the concept. Typically, ADT focuses only on the behaviour. Class, however, is specified in terms of both attributes and behaviour.

6.2.5 Object

It follows that **object** is a realisation of an abstract object in an OOP. That is, for example, every object of the class Customer (in a commercial OOP) corresponds to an

abstract object of CUSTOMER that actually exists in its problem domain. When it is necessary to differentiate between the two terms, we will use the name **concrete object** to refer the objects of an OOP.

We can also use “concrete” to differentiate between two state-specific terms that are discussed in [20]: concrete state and abstract state. **Concrete state** is the state of a concrete object while **abstract state** is the state of an abstract object.

6.3 Design Method

To study class design it is useful to break it down into components and study the structure of each and the relationships between them.

Definition 6.1

*The **design space** of a class consists of three parts: header, state space and behaviour space. **Header** captures the specification the abstract concept that the class represents. **State space** consists in the design of the attributes. **Behaviour space** consists in the design of the object operations.*



The first and primary source of inspiration for the design method in this book is the program specification method presented by Liskov and Guttag [20]. In a previous chapter, we discussed this method for procedural program design. As discussed in [20], the method is also applied to OOP. However, this OOP design method focuses only on the object operations. Although the method discusses object representation (the *rep*), it does not formally capture attributes in the design specification, nor does it describe the transformation steps needed to define the rep from attributes.

Thus, it is necessary to extend the Liskov and Guttag’s method to take into account both attributes and operations in the design specification. This becomes more pressing when we place the method in the broader context of object oriented development that Booch [4] advocated about a decade earlier. A first important extension of the method that is proposed in this book concerns the state space [18]. In particular, the state space incorporates a set of essential design rules (called *domain constraints*) on the attributes. The constraints are expressed directly in OOPL using the annotation construct. These constraints are chosen from the commonly cited constraints in both software engineering and system engineering literatures (e.g. [14, 20]) as well as those used in state-of-the-art software development frameworks (e.g. JPA [24], Hibernate [27], Bean validation [26] etc.).

Another important extension of the OOP design method in this book concerns

the behaviour space [18]. Specifically, the behaviour space uses annotation to express the structural design rules concerning the operation behaviours. These rules are also identified to be essential to the operations that operate on the objects.

In brief, the object oriented design method in this book extends Liskov and Guttag's with three extensions:

- Objects are characterised by both attributes and operations.
- *State space*: attributes have restrictions on values that they can take, these restrictions are formalised as domain constraints. These constraints are expressed directly in OOPL using annotation.
- *Behaviour space*: essential operations are specified with annotations that make explicit their behaviours.

Why using annotation to express constraints?

Annotation is a term borrowed from Java's annotation [11]. The similar term in C# is *attribute*. Conventionally, annotation is used to add additional information to the code elements of the core constructs (class, field, method and parameter). The idea of using annotation in program design is not new. Java itself extensively uses annotation. Annotation is regularly used in Java-based software development tools. These include data management tools (e.g. Java persistence API (JPA), Hibernate, *etc.*) and web-based software development frameworks (e.g. Spring², OpenXava³, *etc.*). Software tools for user interface development, e.g. Angular⁴, also use annotation to express design rules.

Design constraints can be treated as a special type of information that should be added to help strengthen the program model. The main benefits of using annotation to express constraints include:

- making explicit the precise semantics of the program elements
- compile-time validation of program model
- ease of constraint processing (because constraints are written in the same language as the program)

6.3.1 Method Overview

The OOP design method proposed in this book focuses on designing class and consists in the following procedure:

²<https://spring.io/>

³<https://www.openxava.org/>

⁴<https://angular.io/>

1. Specify the class header (Section 6.5)
2. Specify the state space (Sections 6.6 and 6.7)
3. Specify the behaviour space (Section 6.9)

In addition, Section 6.10 discusses how the design method is applied to collection-typed classes. Although the target language used in this book is Java, the design method can easily adapted for other OOPs.

6.4 Design Specification Language

In order to use the design specification language of procedural programs (described an earlier chapter) for OOPs, it is necessary to extend it to support the class structure and some commonly used notations to describe the object state. We will briefly explain both aspects below.

6.4.1 Design Notation

We use two design notations for class design: **UML class diagram** and **design constraint table**.

UML class diagram

As discussed in a previous chapter, a commonly used design notation for class structure is the UML class diagram notation [22]. To ease discuss, we briefly review this notation in Figure 6.3.

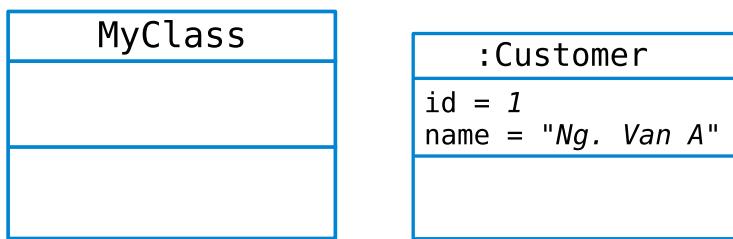


Figure 6.3: Class and object design diagram.

The diagram on the LHS of this figure is a simple **class diagram**, which contains a single class. The top compartment is reserved form class name and, in some cases, special annotations for class. The attributes and operations are placed in the second and third compartments, respectively. The diagram on the RHS of the figure is a simple **object diagram**, which contains a single object of a class named **Customer**.

Design constraint table

Table 6.1 shows the template for the design constraint table, which we use to define the domain constraints of each attribute of a class. The UML class notation does not support the specification of these constraints so they need to be written separately. The table illustrates with the domain constraints of the attribute `Customer.id`. We will look at this and other domain constraints later in this section.

Table 6.1: Design constraint table

Attribute (attribute name)	type (formal type)	mutable (is it mutable (T/F)?)	optional (is it optional (T/F)?)	length (max length)	min (min value)	max (max value)
id	Integer	F	F	-	1	-

6.4.2 Text-Based Set Notation

When designing set abstractions, a set notation is needed to precisely express the state changes that occur as part of the behaviour specification. For example, the operation that inserts a new element to a set will update the set's state such that it contains one more element than the state before the operation is invoked. To express these conditions, this book adopts the text-based set notation presented in [20]. Table 6.2 lists the notation and their textual form used in the specification. Note that the textual forms convert all the superscript and subscript characters to normal ones. In addition, it replaces the set operators by the ASCII characters that best match them; e.g. the union operator \cup is replaced by ‘+’.

Table 6.2: Set notation

Set expressions	Textual form	Description/Example
$\{x_1, \dots, x_n\}$	{x ₁ , ..., x _n }	a set of n elements
$s_1 \cup s_2$	t = s ₁ + s ₂	set union
$s_1 \setminus s_2$	t = s ₁ - s ₂	set difference
$s_1 \cap s_2$	t = s ₁ & s ₂	set intersection
$ s $	s	set cardinality
$x \in s$	x in s	set membership (i.e. x is an element of s)
$\{x \mid p(x)\}$	{x p(x)}	set former (i.e. a set of all elements x such that p(x) is true)

6.4.3 Useful Object Data Types

To design the attributes in a target OOPL requires understanding the data types provided by that language. This section gives an overview of some useful object data

types provided by Java.

Wrapper classes

Wrapper classes, or wrapper types, are classes that 'wrap' the primitive types, making them suitable for use as object types. Thus, wrapper types can be used as both formal and concrete types of attributes. Table 6.3 lists the primitive types and their corresponding wrapper types.

Table 6.3: Wrapper classes.

Primitive types	Wrapper classes
int	Integer
long	Long
float	Float
double	Double
char	Character

A key benefit of using a wrapper class (e.g. Integer) compared to the corresponding primitive data type (e.g. int) is that it supports the specification of null value. This value is needed for integral, optional attributes (i.e. attribute with the property optional =true). It is not possible to initialise this attribute to null if the primitive data type is used.

To ease programming with wrapper objects, Java⁵ supports auto-boxing and auto-unboxing features. The former allows a primitive data value to be automatically converted to an object of the corresponding wrapper type (e.g. int is converted to Integer). The latter allows the opposite conversion from a wrapper object to the corresponding primitive data value.

Example 6.2 Program IntegerWrapper

Listing 6.2 shows a simple IntegerWrapper program that demonstrates the wrapper class Integer and how to use it with the auto-boxing and auto-unboxing features. Similar examples can easily be constructed for other wrapper classes.

```

1 /**
2 * Overview A program that manipulates Integer objects.
3 */
4 public class IntegerWrapper {
5 /**
6 * The run method
7 */

```

⁵starting with Java version 1.5.

```

8  public static void main(String[] args) {
9      Integer i;
10     int j, k;
11
12     // create object using auto-boxing
13     i = 5;                                /* i = Integer(5) */
14
15     // auto-convert to primitive using unboxing
16     k = i;                                /* k = 5 */
17
18     // unboxing i back to primitive in expression
19     j = i + 10;                            /* j = 15 */
20
21     System.out.printf("i, j, k = %d, %d, %d %n", i, j, k);
22 }
23 }
```



Vector as a Dynamic Array

Arrays in Java have a fixed dimension and cannot be resized once created. Practical applications, however, often need to operate on arrays that can automatically grow in size, as elements are added to or removed from them. Class `Vector` (package: `java.util`) is one of oldest classes that provide this functionality. The specification of `Vector` states that it is a class, whose objects store elements of arbitrary types and provide index-based access to these elements. Internally, however, `Vector` uses array to represent elements. Some of the useful operations that `Vector` provides are:

- `add(T o)`: adds element `o` to the vector
- `set(int i, T o)`: replaces the i^{th} element by a new one (`o`)
- `get(int i)`: retrieves the i^{th} element
- `remove(int i)`: removes the i^{th} element from the vector
- `size()`: returns the number of elements in the vector

In addition, `Vector` supports the parameterised (generic) syntax: `Vector<T>`, where `T` is the element type. This is useful if the elements are known to belong to a specific type (e.g. `Integer`). To use `Vector` requires creating a `Vector` object and invokes its operations.

Note that in Java `Vector` is a subtype of the `List` interface, for which there exists two other more well-known concrete subtypes (`ArrayList` and `LinkedList`). This

book uses Vector instead of these other subtypes to design a collection data type, because Vector is the closest as it gets to an array data structure, but is more flexible and easier to work with.

Example 6.3 A vector of integers

Listing 6.1 shows the code of a simple program that demonstrates how to work with Vector. The listing shows how to add primitive integer values to a vector and how auto-unboxing works when retrieving an element and assigning it directly to a primitive-typed variable.

Listing 6.1: Vector<Integer>

```

1 public class IntVector {
2     public static void main(String[] args) {
3         Vector<Integer> v = new Vector<>(); // creates an empty Vector
4         // adds integers to the vector
5         v.add(1); // first element at index 0
6         v.add(1);
7         v.add(2);
8         v.add(3);
9         v.add(5);
10        // prints
11        System.out.printf("Vector: %s%n", v);
12        // retrieve an element at a given index
13        int i = v.get(0); // casting is NOT required
14        // prints
15        System.out.printf("element 0: %d%n", i);
16        // change a particular element
17        v.set (0, -1);
18        // prints
19        System.out.printf("after set(0,-1): %s%n", v);
20        // delete an element at a given index
21        v.remove(0);
22        // prints
23        System.out.printf("after remove(0): %s%n", v);
24    }
25 }
```



Vector or set? Conceptually, vector is similar to set in providing a data structure for storing a collection of items and the items can be of different types. However, there are important differences between these two data structures. First, vector works like array and thus it does not care about whether or not the elements are the same or not. That is, it

is perfectly acceptable to add duplicate elements to a vector (the example in the previous subsection shows this). Sets, however, do not allow duplicate elements. Second, similar to array, vector provides an index-based access to its elements. However, set does not support this type of access (elements in a set are stored in an arbitrary order).

6.5 Header Specification: Specifying the Abstract Concept

6.5.1 Overview

We first define abstract concept in terms of attributes and then use a typical abstract object to clarify the definition. Note that abstract concept does not have behaviours. These are defined for class, which is an object oriented model of the concept. We will discuss the specification of these behaviours in the next section. The concept specification is written in the header comment of the class definition. It is referred to in this note as **header specification**. The specification includes the following components:

- concept name (also used as the class name)
- overview
- attributes
- abstract object
- abstract properties

We will explain the design guidelines for these components in this section. We illustrate our discussion with the two design examples about Customer and IntSet (see Section 6.1).

6.5.2 Choosing a Name

The class is chosen based on the name of the concept that it models. For example, if the concept is CUSTOMER then the class should be named Customer.

Example 6.4 Class name

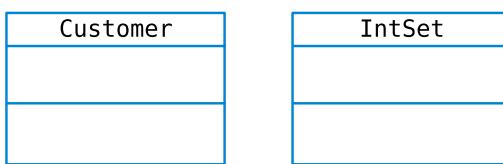


Figure 6.4: Choosing the class name for Customer and IntSet.

Figure 6.4 shows the initial design of Customer and IntSet that show the labelled

class boxes of these two concepts. We will gradually fill details into these boxes as we progress through this chapter. □

6.5.3 Writing the Overview

The overview describes the meaning of the abstract concept. In the UML class diagram, it is written in a note box that is attached to the class box. In the header specification, it is written using tag @overview.

Example 6.5

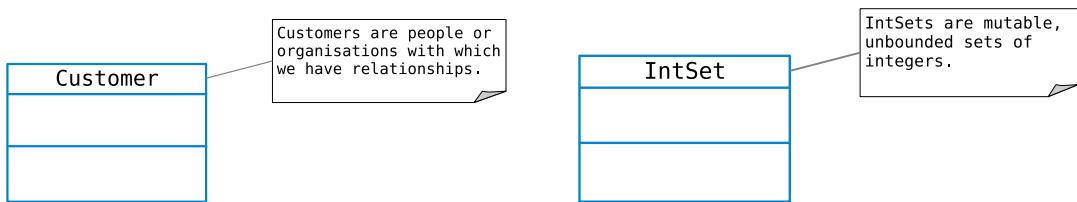


Figure 6.5: Writing @overview in a note box and attaching it to the class.

Figure 6.5 shows the two note boxes in the class diagrams of the two concepts Customer and IntSet. Listing 6.5 lists the header specifications of Customer and IntSet that have been updated with their @overview components. The content of @overview is the same as in the note box on the class diagram.

```

1 /**
2  * @overview Customers are people or organisations
3  *           with which we have relationships.
4 */
5 public class Customer
6
7 /**
8  * @overview IntSet are mutable, unbounded sets of integers.
9 */
10 public class IntSet

```

□

6.5.4 Defining Attributes

The attributes of a concept are drawn in the second compartment of the UML diagram. Note that the attribute visibility (+/-) is not yet determined at this stage. In the header specification, the attributes are written using the tag @attributes. Each attribute entry consists in three parts:

- name: the attribute name

- (formal) type: the abstract data type of the attribute
- concrete type: the actual data type of the attribute (left blank for now, will be added later)

Note that the concrete type will be discussed (and thus updated) later when we explain object representation. The commonly used formal data types are:

- Integer: integral values
- String: text values
- Real: real values (e.g. 1.5, 2.0)
- Char: character
- Boolean: true, false
- Sequence (i.e. array) of the above: e.g. Integer [] is a sequence of integers
- Set of the above: e.g. Set<Integer> is a set of integer

Example 6.6

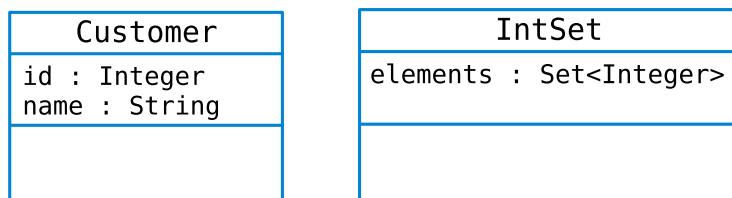


Figure 6.6: Adding attributes to the class design.

Figure 6.6 shows an update to the class diagrams of Customer and IntSet that contain attributes. Listing 6.6 lists the corresponding update to the header specifications of Customer and IntSet that have been updated with their @attributes components. For brevity, we only show in the listing the update content (thus, omitting the @overview content).

```

1 /**
2 * @overview ... as before ...
3 * @attributes
4 *   id   Integer
5 *   name String
6 */
7 public class Customer
8
9 /**
10 * @overview ... as before ...
11 * @attributes
12 *   elements  Set<Integer>
13 */
14 public class IntSet

```



6.5.5 Defining Abstract Object

Recall that one of the quality of design specification is clarity, which is achieved with conciseness combined with a right degree of redundancy. Abstract object is an example of the abstract concept, which is a necessary redundancy that allows the user to understand what the concept is about. We adopt the notation in [20] to write the abstract object, because this notation is simple and concise. Further, it provides the format for the string representation of concrete object, which will later be used to design an object operation. We discuss operation design later in Section 6.9.

The notation for abstract object consists in a statement of the form “A typical C is F, where E”, where C is the class name, S is the string representation of the object and W are expressions describing the elements in S. More specifically, S takes the following format:

1. if C has a single attribute then the typical values of this attribute are listed:
 - if the attribute type is a collection (e.g. set, list, *etc.*) then the general collection notation is used to list the elements e.g. $\{x_1, \dots, x_n\}$ is a typical set object, while $[x_1, \dots, x_n]$ is a typical list or array object
 - otherwise, the typical values of the attribute type are listed; e.g. -2,-1,0,1,2,... are typical values for integer type
2. if the class has several attributes then the tuple notation is used to write the typical values of the attributes.

Example 6.7 Abstract object Figure 6.7 shows object diagrams of the three abstract objects of Customer, IntSet and Integer. Listing 6.7 lists the three abstract object descriptions in the @object tags of the header specifications of the three classes.

For Customer:

- this class has two attributes (`id` and `name`) so the tuple notation is used
- tuple $\langle d, n \rangle$, where `d` and `n` are values of the two attributes (*resp.*)
- attributes `id` and `name` are treated as predicates

For IntSet:

- this class has a single attribute, whose formal type is set-based, so the set notation is used to list its elements

For Integer:

- this class has a single attribute whose type is integral, so its instances are listed

¹ `/**`

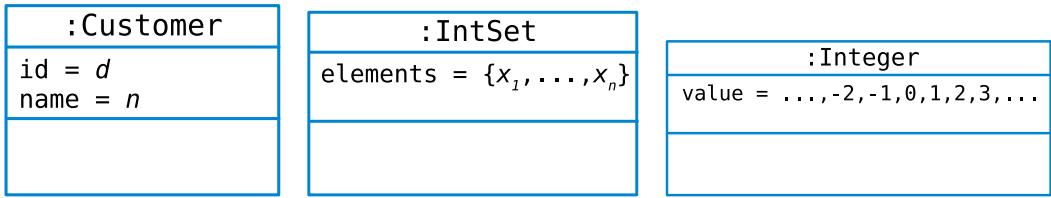


Figure 6.7: Abstract objects of Customer, IntSet and Integer.

```

2 * @overview ... as before ...
3 * @attributes ... as before ...
4 * @object A typical Customer is c=<d,n>, where id(d), name(n).
5 */
6 public class Customer
7
8 /**
9 * @overview ... as before ...
10 * @attributes ... as before ...
11 * @object A typical IntSet object is s = {x1,...,xn},
12 *   where x1,...,xn are elements
13 */
14 public class IntSet
15
16 /**
17 * @overview Integers are immutable whole numbers (incl. 0) and their
18 *   negatives.
19 * @attributes
20 *   value Integer
21 * @object Typical integers are ..., -2, -1, 0, 1, 2, 3, ...
22 */

```

□

6.5.6 Defining Abstract Properties

Overview

An abstract property is a property of the abstract concept. It holds true for all abstract objects. We are interested in a common set of domain properties that are used for concepts. We call these properties *domain constraints*. We use the tag `@abstract_properties` in the header specification to specify all the properties.

Domain constraints

A **domain constraint** is a general statement about what data values an attribute can take. Our analysis of the frequently-used system and software engineering works [14, 20] reveal the following basic constraints:

- **mutable**: whether or not the attribute value can be changed
- **optional**: whether or not the attribute value must be defined for each object
- **length** (primarily used for string type): the maximum length of the values
- **min** (for numeric type): the min value
- **max** (for numeric type): the max value

Each of these properties is defined as a function from the set of attributes into a set of sets of values. The function ranges are as follows:

- **mutable, optional**: range is the set {true, false}
- **length**: range is the set of natural numbers
- **min, max**: range is the set of real numbers

Example 6.8 Domain constraint

Customer Table 6.4 shows the design table for the domain constraints of Customer . Listing 6.2 shows how these properties are written in the `@abstract_properties` section of the header specification.

Table 6.4: Customer's domain constraints.

Attribute	type	mutable	optional	length	min	max
id	Integer	F	F	-	1	-
name	String	T	F	50	-	-

Listing 6.2: Customer

```

1 /**
2 * @overview ... as before ...
3 * @attributes ... as before ...
4 * @object ... as before ...
5 * @abstract_properties
6 *   mutable(id)=false /\ optional(id)=false /\ min(id)=1 /\ 
7 *   mutable(name)=true /\ optional(name)=false /\ 
8 *   length(name)=50
9 */
10 public class Customer

```

IntSet Table 6.5 shows the design table for the domain constraints of IntSet. Listing 6.3 shows how these properties are written in the @abstract_properties section of the header specification.

Table 6.5: IntSet's domain constraints.

Attribute	type	mutable	optional	length	min	max
elements	Set<Integer>	T	F	-	-	-

Listing 6.3: IntSet

```

1 /**
2 * @overview ... as before ...
3 * @attributes ... as before ...
4 * @object ... as before ...
5 * @abstract_properties
6 *   mutable(elements)=true /\ optional(elements)=false /\ 
7 *   elements != {} -> (for all x in elements. x is integer)
8 */
9 public class IntSet

```



Other properties

These are properties that are specific to each abstract concept and are not already covered by the domain constraints. Examples of these properties include:

- elements of a Set are distinct
- elements of an Array form a sequence

Example 6.9 IntSet

Listing 6.4 shows how the distinct property of a set is added to the @abstract_properties specification of IntSet shown in Listing 6.3.

Listing 6.4: IntSet

```

1 /**
2 * @overview ... as before ...
3 * @attributes ... as before ...
4 * @object ... as before ...
5 * @abstract_properties
6 *   mutable(elements)=true /\ optional(elements)=false /\ 
7 *   elements != {} -> (for all x, y in elements. x != y)
8 */
9 public class IntSet

```



6.6 Specifying Concrete Attribute Types

The first important step that needs to be taken before entering the state space design is to determine the concrete attribute type. A **concrete type** is the actual data type in the target language that realises the formal type of an attribute. While the formal type may or may not be available in the target language, the concrete types must be supported by the language. There is usually a one-to-many mapping between the formal and concrete types. One formal type can be realised by one or more concrete types. If the concrete type of an attribute differs from the formal counterpart then it is written in the third column of the attribute entry in the `@attributes` section of the header specification.

The simplest case is when the formal type is supported by the target language (which, in the case of Java, includes all the primitive types and `String`). In this case, the concrete type is the same as the formal counterpart. A more complex situation arises when the formal type is not supported by the target language. In this case, choices can exist among the alternative types provided by the language. For example, suppose the formal type `Set<Integer>` is not directly supported by Java⁶ then two alternative concrete types exist for it: array of integers (`int []`) or `Vector<Integer>`. This raises the question of how can one decide to choose one type over others?

A general criteria suggested by Liskov and Guttag [20] is to choose a type that can **balance between productivity and efficiency**. This balance is desired since types usually do not do well in both. **Productivity** refers to the extent to which it is easier to program with a type. For example, `Vector` is preferred to `array` in terms of productivity for working with dynamic arrays. This is because `Vector` can grow and shrink dynamically and it provides operations for adding and removing the elements. However, `array` is preferred to `Vector` for working with arrays of fixed sizes. **Efficiency**, on the other hand, refers to the run-time efficiency of the code that uses this type. In this aspect, for instance, `array` is preferred to `Vector` since it provides a faster, constant look-up time of the elements.

Guidelines

We summarise below the guidelines for specifying the concrete type of an attribute:

- must be supported by the target language
- may be the same or different from the formal type
- need to balance between productivity and efficiency

⁶In fact, Java has this data type; but, for the sake of the discussion we assume that it does not.

- productivity: ease coding with the type
- efficiency: run-time efficiency of the using code

Example 6.10 Customer

Figure 6.8 and Listing 6.5 show the concrete type design and specification of the two attributes `id` and `name` of `Customer`. For attribute `int`, the primitive type `int` is chosen for efficiency reason. It helps avoid unnecessary boxing and unboxing conversions at run-time. Although Java directly supports the wrapper type `Integer`, the extra operations that this class provides are not needed for client programs that need to manipulate the `id`'s value.

For attribute `name`, the concrete type is the same as the formal one because `String` is directly supported by Java. In addition, class `String` provides many useful operations that make it easier to program with.

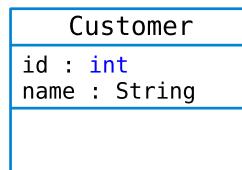


Figure 6.8: Customer with concrete types.

Listing 6.5: Customer

```

1 /**
2 * @overview ...
3 * @attributes
4 *   id   Integer   int
5 *   name String
6 * ...
7 */
8 public class Customer
  
```

□

Example 6.11 IntSet

Figure 6.9 and Listing 6.6 show the concrete type design and specification of the attribute `elements` of `IntSet`. The chosen concrete type is `Vector<Integer>`, although this is not a clear choice in all situations. As discussed above, both `int[]` and `Vector<Integer>` are suitable for use as concrete types. From the productivity perspective, the latter is preferred because `Vector` provides useful operations that helps easily implement most of the essential operations of `IntSet`. As for efficiency, there is no clear winner across all the key operations. In our experiment, `int[]` runs faster with the element removal and membership test operations but is slower with the element

addition operation. Although the client program's code that uses `Vector` is shorter than that using `int []`, `Vector<Integer>` in fact adds another layer of abstraction on top of array, which would impact the overall code performance.

Therefore, if `IntSet` either has a fixed size or does not frequently grow and shrink then `int []` is a preferred concrete type. Otherwise, `Vector<Integer>` should be chosen as the concrete type.

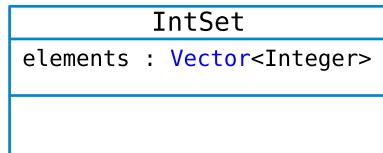


Figure 6.9: `IntSet` with concrete type.

Listing 6.6: `IntSet`

```

1 /**
2 * @overview ...
3 * @attributes
4 *   elements  Set<Integer> Vector<Integer>
5 * ...
6 */
7 public class IntSet
  
```



6.7 Specifying the State Space

State space specification concerns the design of the attributes, which must take into account the constraints associated with them. The attributes and their chosen concrete types form the representation (abbreviated as **rep** [20]) of the objects. The attributes, also called *object* or *instance variables*, represent the sets of values that can be assigned to the object states. In the remainder of this book, where the context is clear we will use the terms ‘attribute’ and ‘instance variable’ interchangeably.

The state space is specified in two steps:

- define an instance variable for each attribute
- annotate each instance variable with suitable domain constraints

The first step is to translate the formal attribute definition in the header specification into suitable instance variables. The second step aims to make the variable definitions precise by translating the abstract properties concerning the attributes into constraints on the corresponding instance variables. This book focuses on translating the domain

constraints into annotations that are attached to the variables. Properties other than the domain constraints can also be translated in a similar manner using new annotations. However, we will not discuss this translation in this book.

6.7.1 Defining instance variables

For each attribute, define an instance variable as follows:

- identifier is the attribute name
- data type is the attribute's concrete type
- access modifier is `private`

The `private` access modifier is necessary to implement the information hiding principle (preventing the variable from being accessed directly by the client program).

Example 6.12 Customer's representation

Figure 6.10 and Listing 6.7 show how the Customer's rep is defined in terms of two instance variables `id` and `name`. The arrow lines in the figure illustrate how the class design specification of the attributes are translated into the instance variables that make up the rep.

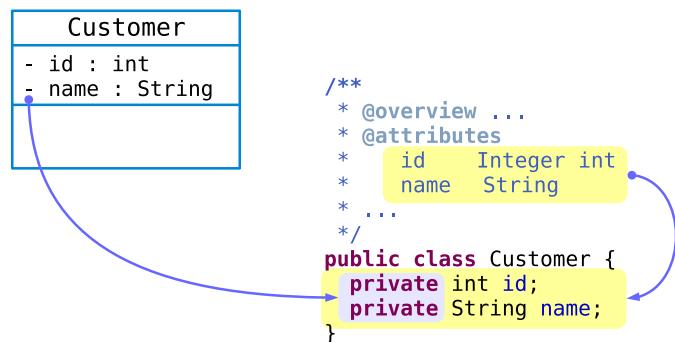


Figure 6.10: Customer's representation.

Listing 6.7: Customer

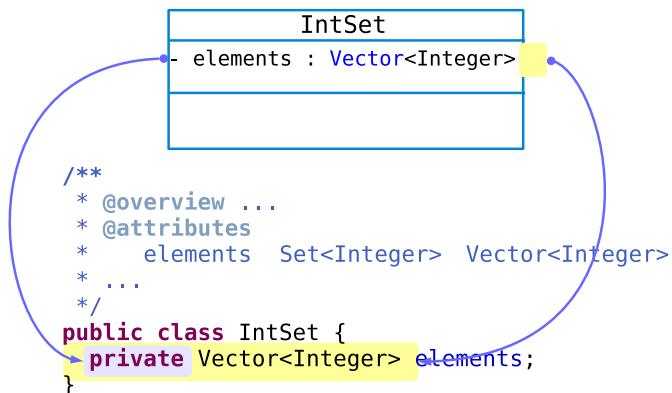
```

1 /**
2  * @overview ...
3  * @attributes
4  *   id Integer int
5  *   name String
6  * @object ...
7  * @abstract_properties ...
8 */
9 public class Customer {
10   private int id;
11   private String name;
12 }
  
```

□

Example 6.13 IntSet representation

Figure 6.11 and Listing 6.8 show how the IntSet's rep is defined in terms of the instance variable `elements`. Note that the choice of `Vector<Integer>` helps guarantee the abstract property that IntSet only contains integers.

**Figure 6.11:** IntSet's representation.**Listing 6.8:** IntSet

```

1 /**
2  * @overview ...
3  * @attributes
4  *   elements Set<Integer> Vector<Integer>
5  * @object ...
6  * @abstract_properties ...
7 */
8 public class IntSet {
9   private Vector<Integer> elements;
10 }
```

□

6.7.2 Annotating instance variables with domain constraints

The essential domain constraints defined in Section 6.5.6 are realised by the annotation `@DomainConstraint` shown in Listing 6.9. This annotation is defined in a package named `utils`, and thus will need to be imported into each class before usage.

Listing 6.9: Annotation `@DomainConstraint`

```

1 @Retention(RetentionPolicy.RUNTIME)
2 public @interface DomainConstraint {
3   public String type() default "null";
4   public boolean mutable() default true;
```

```

5   public boolean optional() default true;
6   public int length() default -1;
7   public double min() default Double.NaN;
8   public double max() default Double.NaN;
9 }
```

As shown in the listing, each domain constraint is mapped to an annotation element of the same name. For example, the `mutable` constraint is mapped to the element `DomainConstraint.mutable`. All annotation elements are given a default value and thus can be omitted if not applicable to the attribute. Annotating an instance variable with `@DomainConstraint` involves assigning the annotation's elements to some suitable values. These values are taken from the corresponding domain constraint rules in the `@abstract_properties` section.

Example 6.14 Customer's domain constraints

Figure 6.12 and Listing 6.10 show how the two instance variables of class `Customer` are annotated with two `@DomainConstraints`, realising the abstract properties of the two corresponding attributes. The arrow lines in the figure illustrate the mapping between the abstract properties and the annotation elements.

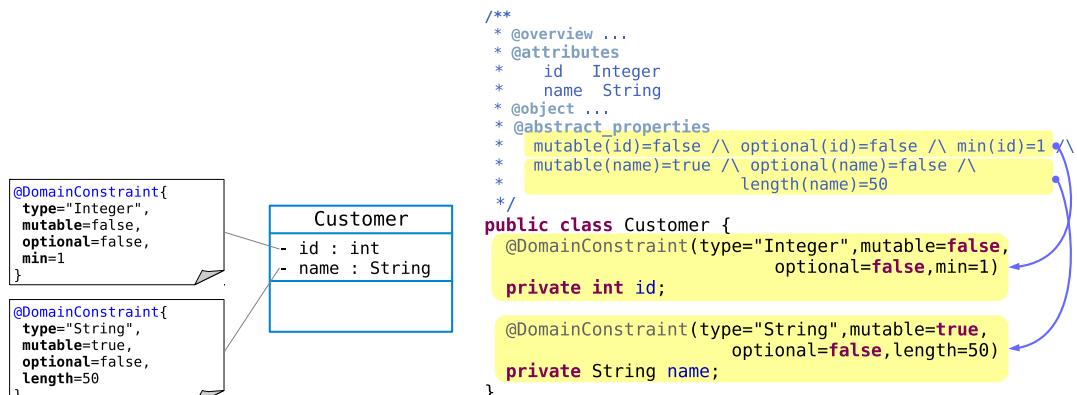


Figure 6.12: Customer representation with constraints.

Listing 6.10: Customer

```

1 /**
2  * @overview ...
3  * @attributes
4  *   id Integer
5  *   name String
6  * @object ...
7  * @abstract_properties
8  *   mutable(id)=false /\ optional(id)=false /\ min(id)=1 /\ 
9  *   mutable(name)=true /\ optional(name)=false /\ length(name)=50
10 */

```

```

11 public class Customer {
12     @DomainConstraint(mutable=false,optional=false,min=1)
13     private int id;
14
15     @DomainConstraint(mutable=true,optional=false,length=50)
16     private String name;
17 }

```

□

Example 6.15 IntSet's domain constraints

Figure 6.13 and Listing 6.11 show how the instance variable `elements` of class `IntSet` is annotated with a `@DomainConstraint` that realises the abstract properties of the corresponding attribute. Note that the annotation only captures the domain constraints, it does not (yet) support the uniqueness property of `IntSet`. This property is checked by a special validation operation, which we will discuss later in Section 6.9.

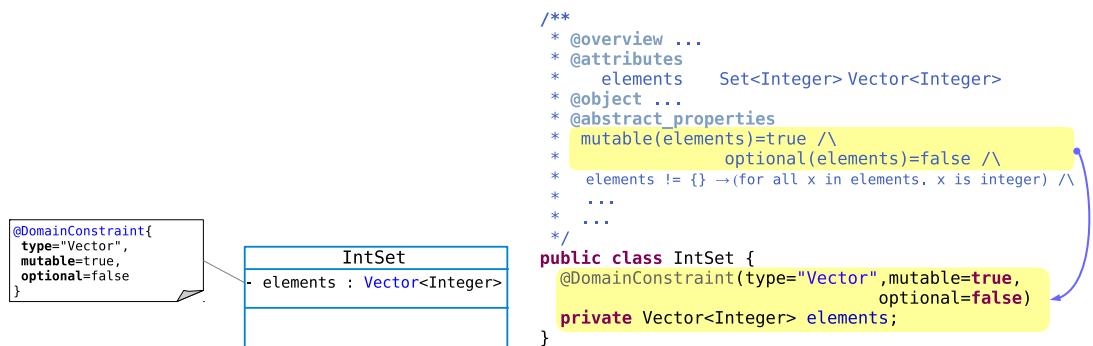


Figure 6.13: `IntSet` representation with constraints.

Listing 6.11: `IntSet`

```

1 /**
2  * @overview ...
3  * @attributes
4  *   - elements Set<Integer> Vector<Integer>
5  * @object ...
6  * @abstract_properties
7  *   mutable(elements)=true /\ optional(elements)=false /\ ...
8  *   elements != {} -> (for all x in elements. x is integer) /\ ...
9  *   elements != {} -> (for all x, y in elements. x != y)
10 */
11 public class IntSet {
12     @DomainConstraint(mutable=true,optional=false)
13     private Vector<Integer> elements;
14 }

```

□

6.8 Essential Design Annotations

Before discussing the next part of the design, let us discuss in this section the set of essential design annotations that are used in class design. These annotations are a unique contribution of this book.

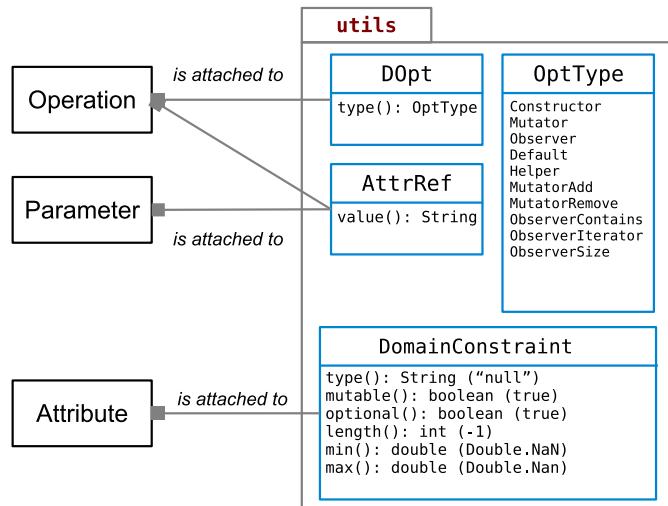


Figure 6.14: The design annotations.

The ultimate aim of the annotations is to make class design more precise. The essential annotations (shown in Figure 6.14) are defined in the package named `utils` of the attached source code. Annotation `@DomainConstraint` was discussed in the previous section. Other annotations are used to annotate object operations, which we will discuss in the next section. The precise definition of each annotation will be given when they are used in the text.

Conceptually, the operational annotations help make explicit the behaviour of the operations in relation to the attributes whose values they manipulate. For example, attribute `Customer.name` is manipulated by two operations `setName` (changing the value) and `getName` (accessing the value). To make explicit this connection requires annotating the two operations with a reference to the attribute.

What if an OOPL does not Support Annotations?

Although the well-known OOPLs (e.g. Java, C#) support the annotation construct, it is not the case for all OOPLs. There are two design alternatives for languages that do not support this construct:

1. if there is an alternative construct then translate the annotations into it
2. otherwise, leave the annotations in the code but comment them out. The commented annotations will serve as valuable documentation for the code

6.9 Specifying the Behaviour Space

Behaviour space specification concerns the design of object operations. Operations (a.k.a object procedures) typically observe and/or modify the attribute values that make up the object state. Two key design questions are (1) what are the essential operations? and (2) how to specify these operations. To answer the first question, we discuss a basic classification of the operation types. To answer the second question, we adapt the procedural design specification approach to design operations. A key difference here is to ensure that the behaviours of operations preserve the abstract properties.

6.9.1 Essential Operation Types

Liskov and Guttag [20] gives a classification which include four essential operation types. This book adds two operation types (default and helper) to this classification. We briefly describe the operation types below. A more detailed discussion of each operation type will be given in later sections.

- **Creator** (a.k.a constructor): creates objects of a class from attribute values
- **Producer**: creates a new object from an existing object of the same type
- **Mutator**: changes the object state by modifying the attribute value(s)
- **Observer**: obtain information about object state (through the attribute values)
- **Default**: a default operation for object types provided by the target language
- **Helper**: a utility operation that performs a task needed by other operations

6.9.2 How to Choose the Operations?

Although there are only six essential operation types, there are potentially an infinite number of operations that belong to each one. It is therefore helpful to determine what constitute the essential operations of each type for a class. The general guidelines stated by Liskov and Guttag [20] are as follows:

- consider the usage context:
 - e.g. operation `IntSet.size` is needed if the cardinality of set is required
- define at least one constructor and one observer
 - In Java, the default (non-parameter) constructor may be omitted
- define a mutator for each mutable attribute
- define data validation operations for attributes with domain constraints
- define operations that can help ease programming with objects:
 - e.g. define `IntSet.isIn` to check if a set has a specific element

- define helper operations to help other operations:

- e.g. `IntSet.getIndex`

Example 6.16 Customer & IntSet

Figures 6.15 and 6.16 respectively show the essential operations of the two classes `Customer` and `IntSet`. The reasons for choosing these operations will be given in the subsequent sections. \square

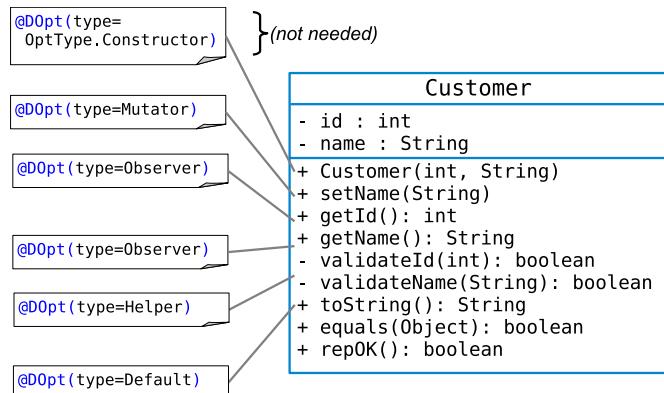


Figure 6.15: Customer operations.

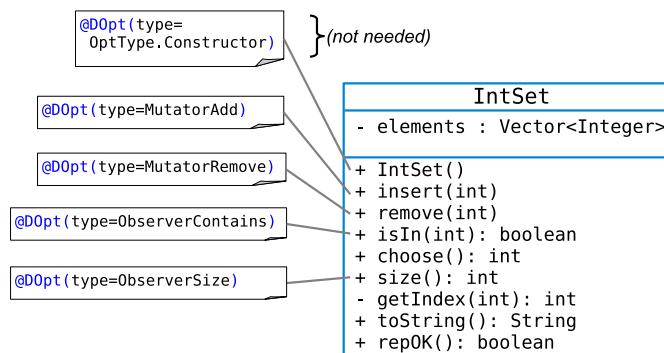


Figure 6.16: IntSet operations.

6.9.3 General Specification Guidelines

The specification method for the operation is adapted from the procedural design specification method that we explained in an earlier chapter. The general specification guidelines are summarised below. More specific guidelines for each operation type will be given in later sections.

- define the operation header *without* the keyword `static`
- annotate an operation suitable `DOpt`, `AttrRef` annotations
- write the `@effects` clause with consideration to the attributes & abstract properties:

- use keyword `this` to refer to the current object and its attributes or operations, e.g. `this.id`, `this.validateId()`, etc.
- use postfix ‘`_post`’ to denote value of a variable in the post-condition [20]
- if domain constraint is available then validate data first
- consider the domain-specific abstract properties, e.g. elements of set must be distinct

6.9.4 Annotation Guidelines

Two operation annotations that are used to design the operations are `DOpt` and `AttrRef`.

Annotation `DOpt`

This annotation describes a behaviour pattern. It is attached to operation to specify that the operation’s behaviour has a certain pattern. `@DOpt` has one property named `type`, which specifies the operation type. The data type of this property is an enum called `OptType`. The annotation is written before the operation header as follows:

`@DOpt(type=t)`, where $t \in \text{OptType}$.

Note that `DOpt` is usually *not* required for constructor and helper operations. The following listing shows the definitions of the essential operation type constants that are defined by the enum `OptType`. The comments in the listing briefly describe the behaviour of each operation type.

```
public enum OptType {
    /**mutator operation (e.g. setter) */
    Mutator,
    /**observer operation (e.g. getter)*/
    Observer,
    /**a specialised type of observer, which is used to annotate an
     *  operation of a collection class that realises its primary
     *  iterator abstraction */
    ObserverIterator,
    /**a specialised type of observer, which is used to annotate an
     *  operation of a collection class that checks for membership
     *  of an element in a collection */
    ObserverContains,
}
```

```

/**a specialised type of observer, which is used to annotate an
operation of a collection class that returns its size*/
ObserverSize,
/**a specialised type of mutator, which is used to annotate an
operation of a collection class that adds a new element to
the collection*/
MutatorAdd,
/**a specialised type of mutator, which is used to annotate an
operation of a collection class that removes an element from
the collection*/
MutatorRemove
;
}

```

The first two operation types are generic types that are used to annotate the mutator and observer operations. The remaining five operation types are specifically used to annotate operations of the collection-typed classes (e.g. IntSet). Annotations MutatorAdd and MutatorRemove, in particular, are special forms of Mutator. Similarly, ObserverContains, ObserverSize and ObserverIterator are the special forms of Observer. We will not discuss operation ObserverIterator in this book.

Annotation AttrRef

This annotation has one property named value, which specifies the name of the attribute that is manipulated by (and thus referenced in) the operation's behaviour. The data type of this property is String. This annotation can annotate both operation and operation parameter and is written with the following syntax:

`@AttrRef(value=n)` or simply `@AttrRef(n)`, where n is the referenced attribute's name.

Note the followings about `@AttrRef`'s usage:

- `AttrRef` is *often* (but not always) used with `DOpt`.
- for *non-constructor operations*: `AttrRef` is used to annotate the operation (not its parameter).
- for *constructor operations*: `AttrRef` is used to annotate the parameters and is written immediately before the declaration of each parameter.
(`AttrRef` is usually *not* required for parameters of non-constructor operations).

6.9.5 Constructor

Let us now begin looking at the design specification of each operation type. This section starts with the constructor operation. Objects need to be created before they can be used in the program. Recall that the purpose of constructor is to perform this task. There are two main methods for defining constructors. The first method is to define a default constructor that has no parameters. Objects created with this constructor have a default state, where the attributes are initialised to the default values. After creation, the desired object state needs to be set by using the mutator operations. The second method is to define an **essential constructor** that takes the essential parameters needed to initialise the attributes. Objects created with this method have the desired state at creation time and, as such, there is no need to use the mutator operations to update it. In practice, both methods are often used which result in a class having several constructors defined.

While the first method is straightforward and is directly supported by the mutator operations defined in this book, the second method is essential to help ensure that the created objects are valid objects (a.k.a *correct by creation*). This section will focus on explaining the specification guidelines for the essential constructor. In particular, the specification will use annotation `@AttrRef` to map the constructor parameters to the corresponding attributes.

Header

In addition to the standard OOP's syntax rules for constructor, the parameter list of the constructor needs to make explicit the essential parameters. An **essential parameter** is a parameter that is defined based on a non-optional, non-collection-typed attribute. Such an attribute has `@DomainConstraint(optional=false)` and a non-collection data type.

The header rules of essential constructor are summarised as follows:

- operation name is same as the class
- return type is omitted
- parameter list contains all the essential parameters
- each parameter is annotated with an `@AttrRef` referencing the name of the corresponding attribute
- each parameter type must match the corresponding attribute's type

Behaviour

Constructor is usually the total operation (i.e. has no `@requires` clause). The `@effects` clause should state the followings:

- initialise an object with the input parameters
- validate input parameters against the domain constraints (if any). If a violation occurs then terminates immediately with an error object of the type `NotPossibleException`

Terminating an operation with an error object is technically called *throwing an exception*. In this case, the exception (or error) object is an object of the `NotPossibleException` class, which is provided in the `utils` package of the attached source code. This object is created with a `String` input argument that captures the error message. The following example will demonstrate how to specify the constructor to throw an exception.

Example 6.17 Customer

Listing 6.12 shows the design specification of the essential constructor of class `Customer`. Note the followings about this specification:

- the header declares to throw `NotPossibleException`
- two essential parameters are needed for two non-optional, non-collection-typed attributes: `id` and `name`
- each essential parameter is attached with an `@AttrRef`, specifying the mapped attribute's name
- `@effects` clause states data validation for both parameters because domain constraints are defined for both attributes. It also details the condition under which the `NotPossibleException` is thrown.

Listing 6.12: Customer

```

1 /**
2 * @effects
3 *   if custID, name are valid
4 *     initialise this as <custID,name>
5 *   else
6 *     throws NotPossibleException
7 */
8 public Customer(@AttrRef("id") int custID, @AttrRef("name") String name)
9      throws NotPossibleException

```



6.9.6 Mutator

Mutator operations are to update the object state. This is usually performed by changing the attribute values. This explains why a most common form of mutator operations are called **setters**, which directly sets an attribute's value to a new one. In this section, we will focus on design specification of this type of operation.

An important rule which must be observed is that setter operations are required for mutable attributes and forbidden for immutable ones.

Header

In addition to the standard OOPL syntax for operation, the following header rules need to be observed for the setter of an attribute named *A*:

- is annotated with `@DOpt.type=OptType.Mutator` and `@AttrRef.value=A`
- operation name has the pattern `setA`
- return type is `boolean`, which signifies whether or not a behaviour invocation succeeds (see behaviour description below)
- parameter list has one parameter whose type matches *A*'s type

Behaviour

Mutator is a total operation, whose behaviour description must be as follows:

- `@modifies` clause states that the object state is changed
- `@effects` clause states the update of the referenced attribute and the handling of input errors by returning `false`

The client program must check the return value to act accordingly. Note that an alternative technique for handling the input errors is to throw an exception (similar to the constructor's design). However, this technique will not be used for mutators in this book.

Example 6.18 Customer

Listing 6.13 shows the design of the setter `setName`, which updates the value of attribute `Customer.name`. Note that there must not be a setter for attribute `id` because this attribute is immutable. If the parameter name is valid and attribute name is set to its value then the operation returns `true`. Otherwise, the operation returns `false`.

Listing 6.13: Customer

```

1 /**
2 * @modifies this
3 * @effects

```

```

4  *   if name is valid
5  *     set this.name to name
6  *     return true
7  *   else
8  *     return false
9  */
10 @DOpt(type=OptType.Mutator) @AttrRef("name")
11 public boolean setName(String name)

```



6.9.7 Observer

Observers obtain information about the object state. A most common type of observer is called **getter**, which directly observes (i.e. gets) the value of an attribute. Because getter operations generally useful, every class should be defined with one getter operation for each attribute. This section will focus on the design specification of getter operations.

Header

In addition to the standard OOPL syntax for operation, the following header rules need to be observed for the getter of an attribute named A :

- is annotated with `@DOpt.type=OptType.Observer` and `@AttrRef.value=A`
- operation name has the pattern `getA`
- return type matches the A 's data type
- parameter list is empty

Behaviour

The behaviour description of getter is very simple, which contains an `@effects` clause stating the result to be equal to the target attribute.

Example 6.19 Customer

Listing 6.14 shows the design of two getters `getId` and `getName` of the class `Customer`. These operations observe the attributes `id` and `name`, respectively.

Listing 6.14: Customer

```

1 /**
2  * @effects return id
3 */
4 @DOpt(type=OptType.Observer) @AttrRef("id")

```

```

5 public int getId()
6
7 /**
8 * @effects return name
9 */
10 @DOpt(type=OptType.Observer) @AttrRef("name")
11 public String getName()

```



6.9.8 Default

Default operations are operations that are provided by the OOPL for every class. They are inherited from a generic `Object` class that serves as the template for all class definitions. Although this fact is hidden from the class designer, the operations are always available for use in each class when needed.

In Java, three common default operations that should be defined for each class are: `toString`, `equals` and `hashCode`. Although our design specification of these operations is written for Java, the design guidelines are also applicable to other OOPLs. A general header rule is that each of these operations should be annotated with the built-in annotation `@Override`. This makes explicit the fact that the operation changes (or overrides) the behaviour of the base operation in class `Object`. The behaviour description needs not be specified, because it should conform to the behaviour of the base operation. However, it is beneficial to write a brief behaviour description explaining how each operation specialises the base one.

`toString`

This operation has an empty parameter list and returns a string representation of the object. This string is formatted based on the details specified in the `@Object` section of the class header specification. The Java header for overriding operation `toString` is as follows:

```

1 @Override
2 public String toString()

```

`equals`

This operation takes an `Object`-typed parameter and returns a `boolean`, indicating whether or not the input object is equal to the current object. The meaning of object

equality is that two objects are “equal” if and only if their behaviours are equivalent. The behaviour description of this method in Java makes this clear by stating the followings: “The equals method implements an equivalence relation on non-null object references” [23]. The Java header for overriding operation `equals` is as follows:

```
1 @Override
2 public boolean equals(Object o)
```

The default Java implementation of this operation is the most restrictive one, which compares two objects by their built-in object identifiers. A more practical (less restrictive) implementation would compare objects by the (mostly) immutable parts of the object state.

`hashCode`

This operation takes no parameters and return an `int` hash value of the object. This value is used as key in a hash data structure for storing the object. Common examples of this type of data structure include `Hashtable` and `HashMap` – both are provided by Java. The Java header for overriding operation `hashCode` is as follows:

```
1 @Override
2 public int hashCode()
```

Behaviourally, this operation must be consistent with `equals`. That is, “... if two objects are equal according to the `equals(Object)` method then calling the `hashCode` method on each of the two objects must produce the same integer result.” [23]. Thus, similar to `equals`, the parts of the object state that are used to compute the hash code should be (mostly) immutable.

6.9.9 Helper

A helper operation performs a behaviour that is used by some other operations or that helps make the objects more useful. Among the commonly useful helper operations are the followings:

- `repOK`, which is short for “representation OK” [20]
- Data validation
- Utility

repOK

The primary purpose of this operation is to check the current object representation to ensure that it satisfies the abstract properties. The idea is that if this check is passed with every object of the class then the object representation is valid.

Another usage of this operation is to test the validity of an object state against its abstract properties. Given that the object representation is correct, an object state is valid (*w.r.t* to this representation) if all the attribute values satisfy the constraints defined in the abstract properties. Thus, this operation can be used to identify (and ignore) the invalid objects in a program. The Java header definition for this operation is as follows:

```

1 /**
2 * @effects
3 *   if this satisfies abstract properties
4 *   return true
5 * else
6 *   return false
7 */
8 public boolean repOK()

```

Data Validation

The purpose of data validation operation is to check that some input data, which are set for an attribute, are valid *w.r.t* the attribute's domain properties. Clearly, if the result is "No" then the data must not be used to set the attribute. This operation is invoked by all operations (e.g. constructor, setter, repOK)) that accept and process the input in some way. Here are the header design guidelines for a data validation operation of an attribute named *A*:

- operation name has the pattern validate*A*
- access modifier: *private*
- return type is *boolean*
- parameter list contains one parameter, whose type must match the attribute's data type

The standard design specification template for this operation should look like this:

```

1 /**
2 * @effects
3 *   if A is valid
4 *   return true
5 * else

```

```

6 *      return false
7 */
8 private boolean validateA(int p)

```

Note that the `@effects` does not need to state specifically what the constraints concerning attribute A are. The statement “ A is valid” implies that this it is valid w.r.t to the constraint properties. A data validation operation of the same class can invoke some other data validation operations, if necessary.

Example 6.20 Customer

Listing 6.15 shows the design specification of two data validation of class `Customer`. These operations validate the input value for the attributes `id` and `name`

Listing 6.15: Customer

```

1 /**
2 * @effects
3 *   if id is valid
4 *     return true
5 *   else
6 *     return false
7 */
8 private boolean validateId(int id)
9
10 /**
11 * @effects
12 *   if name is valid
13 *     return true
14 *   else
15 *     return false
16 */
17 private boolean validateName(String name)

```



Utility

Utility operations are added to ease the implementation of other operations. They are determined from the specifications of the existing operations. If the operations are useful for the general use of objects then they can be defined as `public`, otherwise they are defined as `private`.

An example of a public utility operation is the operation `IntSet.isIn`, which checks membership of elements of an integer set. Internally, this method is invoked by

two other essential operations of the class, namely `insert` and `remove`. Externally, it is invoked by the client program to check for set membership.

An example a private utility operation is the operation `reduce` of a class `Rat` [20], which represents rational numbers. This operation reduces a rational number to its basic form, where the numerator and denominator have no common divisors (e.g. $\frac{3}{6}$ is reduced to $\frac{1}{2}$). This operation is invoked internally by operation `equals` to check equality of two irrational numbers.

Example 6.21 IntSet

Listing 6.21 shows the design specification of a private utility operation, named `getIndex`, of class `IntSet`. This operation is used by both `insert` and `remove` to determine whether an element is in the set. Although this can also be achieved by `isIn`, operation `getIndex` can also return the actual index of an element if it is a member of the set. Operation `remove` uses this index to remove the element.

```

1 /**
2 * @effects
3 *   if x is in this
4 *     return the index where x appears
5 *   else
6 *     return -1
7 */
8 private int getIndex(int x)

```

Thus, all three operations `isIn`, `insert` and `remove` of `IntSet` use operation `getIndex` to carry out their behaviours. Note that, unlike `isIn`, operation `getIndex` must **not** be made public. The main reason is because doing so will make the element index publicly available, despite that this is not a set property. \square

6.10 Specifying the Collection Classes

We discussed in Section 6.9.4 some operation types that are used for collection classes. In this section, we will discuss them in detail. Collection classes differ from non-collection classes in that their operations do not usually follow the set-get pairing discussed earlier. The mutator and observer operations are primarily designed to add/remove/observe one or some element(s) at a time, not all elements at once.

This is indeed demonstrated in the Java's built-in `Collection`⁷ type (called an *interface*) that is used as template for defining all collection classes. This type defines

⁷located in the package `java.util`.

the following operations that are common to all types of collection:

- `add(E e)`: adds an element `e` to the collection
- `remove(E e)`: removes an element `e` from the collection
- `contains(E e)`: determines whether or not the collection contains an element `e`
- `size`: returns the number of elements in the collection

Therefore, it is necessary to specialise the mutator and observer operations for the collection operations. In this book, the aforementioned four basic operations are annotated with the following four OptTypes (*resp.*): `MutatorAdd`, `MutatorRemove`, `ObserverContains` and `ObserverSize`.

6.10.1 Collection Class Marker

In addition to the specialised OptTypes, it is necessary to mark each collection class as a “collection type”. This book adopts the Java’s approach to introduce an interface for this, named `Collection`. This interface is located in the package `utils.collections` of the attached source code. However, unlike the Java’s `Collection` interface, the proposed interface does not force the implementing class to implement any operations. It is up to the class designer to decide which operations the class needs to implement. In the remainder of this chapter, the name `Collection` will refer to the new interface, not the Java’s one.

For example, Listing 6.16 shows the header definition of the class `IntSet`, which implements the `Collection` interface.

Listing 6.16: IntSet

```

1 import utils.collections.Collection;
2 /**
3 * ...
4 */
5 public class IntSet implements Collection {
6     //
7 }
```

6.10.2 Using DOpt with Specialised Types

As discussed above, the essential collection-specific operations are annotated with `@DOpt` whose types are set to one of the four new specialised OptTypes. In the essential design, a collection class has one main attribute which keeps the elements. Let us name this the `elements` attribute. All the essential operations distinctively update or observe

this attribute's value in some way. Thus, there is no need to annotate the operations with the annotation. However, if the collection class is extended with other attributes then it would be necessary to annotate these operations with `@AttrRef`.

Let us briefly discuss the essential design guidelines for three key operation types of collection class.

6.10.3 Constructor

In the essential design, collection objects are created to be empty and are later updated with elements via the mutator operations. Thus, the **essential constructor** of collection class is the constructor that has an empty parameter list. The header of this constructor is the same as the default constructor, but its behaviour must state to initialise an empty object. This involves initialising the `elements` attribute to be an empty array or an empty collection object.

Example 6.22 IntSet

Listing 6.17 shows the design specification of the essential constructor of `IntSet`.

Listing 6.17: `IntSet`

```

1 /**
2  * @effects initialise this to be empty
3 */
4 public IntSet()

```



6.10.4 Mutator

Two essential operations `MutatorAdd` and `MutatorRemove` are responsible for updating a collection with elements. Both operations must preserve the abstract properties of the collection. For example, operation `MutatorRemove` must not change the collection if the input element is not present in the collection. As for operation `MutatorAdd`, it must not add the input element if the collection is a set and the element is already a member of the collection.

Example 6.23 IntSet

Listing 6.18 shows the design specifications of `MutatorAdd` and `MutatorRemove` operations of class `IntSet`. Both operations contain a `@modifies` clause which states that the current collection is updated. The `@effects` clause uses the notation `this_post` to refer to the current `IntSet` object after it has been modified.

Listing 6.18: `IntSet`

```

1 /**
2  * @modifies this
3  * @effects
4  * if x already in this
5  *   return false
6  * else
7  *   add x to this, i.e., this_post=this+{x}
8  *   return true
9 */
10 @DOpt(type=OptType.MutatorAdd)
11 public boolean insert(int x)
12
13 /**
14  * @modifies this
15  * @effects
16  * if x is not in this
17  *   return false
18  * else
19  *   remove x from this, i.e. this_post=this-{x}
20  *   return true
21 */
22 @DOpt(type=OptType.MutatorRemove)
23 public boolean remove(int x)

```



6.10.5 Observer

As discussed earlier in this section, two essential observer operations `ObserverContains` and `ObserverSize` respectively provide information about the membership of an element and the number of elements in the collection. Another essential observer that is also commonly found in a collection class is one that returns the elements in some fashion. In this book, we will consider a most basic type of such operation. It is a getter operation that returns an array of the elements. This operation is marked as an `Observer`. Again, there is no need to annotate the operation with `@AttrRef`, because it operates on the `elements` attribute.

Besides the above operations, other observers may be added depending on the specific need of the collection type.

Example 6.24 IntSet

Listing 6.19 shows the design specification of four observer operations of class

IntSet. The first three operations are the essential operations of the class. The fourth operation is an example of a useful observer operation for IntSet. It returns an arbitrarily element of the set.

Listing 6.19: IntSet

```

1  /**
2   * @effects
3   *   if x is in this
4   *     return true
5   *   else
6   *     return false
7   */
8  @DOpt(type=OptType.ObserverContains)
9  public boolean isIn(int x)
10
11 /**
12  * @effects return the cardinality of this
13 */
14 @DOpt(type=OptType.ObserverSize)
15 public int size()
16
17 /**
18  * @effects
19  * if this is not empty
20  *   return array Integer[] of elements of this
21  * else
22  *   return null
23 */
24 @DOpt(type=OptType.Observer)
25 public Integer[] getElements()
26
27 /**
28  * @effects
29  * if this is empty
30  *   throw an IllegalStateException
31  * else
32  *   return an arbitrary element of this
33 */
34 @DOpt(type=OptType.Observer)
35 public int choose() throws IllegalStateException

```



Chapter 6 Exercise

The exercises in this chapter help practise class design with UML class diagram notation and design specification. To ease reading, some of the relevant exercises given in [20] are reproduced here.

- (Greeting Conversation)** Figure 6.17 is an extended design diagram of the greeting conversation program discussed in the previous tutorial. This diagram shows a few more attributes of the two classes Person and MobilePhone, but it presents only a partial list of operations.

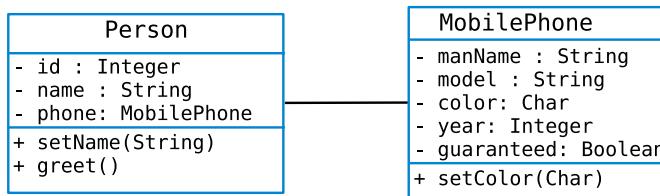


Figure 6.17: An initial design diagram for the greeting conversation program.

The following table is a partially completed table of the domain constraints that apply to the attributes of the two classes in Figure 6.17. Note that the “type” column lists the formal types of the attributes. Attribute `MobilePhone.color` takes a character value that denotes a colour of the phones. The characters are: ‘R’ (for red), ‘O’ (for orange), ‘Y’ (for yellow), ‘B’ (for blue), ‘P’ (for purple). Attribute `MobilePhone.guaranteed` takes the value `true` for a mobile phone if this phone has a guarantee; it takes the value `false` if otherwise.

Class	Attribute	type	mutable	optional	length	min	max
Person	id	Integer	F	F	-	1	-
	name	String	T	F	30	-	-
	phone	MobilePhone	T	T	-	-	-
MobilePhone	manName	String					
	model	String					
	color	Character					
	year	Integer					
	guaranteed	Boolean					

- Complete the domain constraints in the table, using your practical understanding of the application.
 - Write the header and state space specifications for each class.
 - Determine a minimum set of operations needed for each class. Justify your choice of each operation.
 - Write the behaviour space specification for each class.
- (Greeting conversation v1.1)** Implement an enum named Color that captures the different colours mentioned in the program requirement. Update class MobilePhone

to use this enum.

3. **(Greeting conversation v1.2)** Update class `MobilePhone` to address an additional constraint that attribute `model` must be of the form `M-ABC-MNP`, where `ABC` is a 3-letter word and `MNP` is a 3-digit word. For example, `M-SAM-123` is a valid phone model, but `M-SOM-123` is not.
4. **(Greeting conversation v1.3)** Update class `Person` to address an additional constraint that attribute `name` must consist of at least two words that are separated by a white space.
5. Specify a class `EvenIntSet` that represents a set of even numbers. This is an integer set that only accepts even numbers as elements.
6. (see [20]) Specify a map class, named `StringIntMap`, which maps strings to integers. Maps allow an existing mapping to be looked up. Maps are also mutable: new pairs can be added to a map, and an existing mapping can be removed. Be sure that your data type is adequate.
7. (see [20]) Specify a class `IntQueue` that represents a bounded queue of integers. A bounded queue is a queue that has an upper bound, established when the queue is created, on the number of integers that can be stored in the queue. Queues are mutable and provide access to their elements in first-in/first-out order. `IntQueue` operations include:

```
IntQueue(int n)
enq(int x)
int deq()
```

The constructor creates a new queue with maximum size `n`, `enq` adds an element to the front of the queue, and `deq` removes the element from the end of the queue. You may include extra operations as needed for adequacy.

8. (see [20]) Specify a rational number type, named `Rat`.

6.A Customer

The complete specification of Customer.

```
1 import utils.AttrRef;
2 import utils.DOpt;
3 import utils.DomainConstraint;
4 import utils.NotPossibleException;
5 import utils.OptType;
6 /**
7 * Overview Customers are people or organisations with which we have
8 * relationships.
9 * @attributes
10 * id Integer int
11 * name String
12 * @object A typical Customer is c=<d,n>, where id(d), name(n).
13 * @abstract_properties
14 * mutable(id)=false /\ optional(id)=false /\ min(id)=1 /\
15 * mutable(name)=true /\ optional(name)=false /\ length(name)=50
16 * @author dmle
17 */
18 public Customer {
19     @DomainConstraint(optional = false, optional = false, min = MIN_ID)
20     private int id;
21
22     @DomainConstraint(optional = false, length = LENGTH_NAME)
23     private String name;
24
25     // constants
26     private static final int MIN_ID = 1;
27     private static final int LENGTH_NAME = 50;
28
29     /**
30     * @effects
31     * if custID, name are valid
32     * initialise this as <custID,name>
33     * else
34     * throws NotPossibleException
35     */
36     public Customer(@AttrRef("id") int custID, @AttrRef("name") String name)
37         throws NotPossibleException
38
39     /**
40     */
41 }
```

```
38     * @effects
39     *   if name is valid
40     *     set this.name to name
41     *     return true
42     *   else
43     *     return false
44     */
45 @DOpt(type=OptType.Mutator) @AttrRef("name")
46 public boolean setName(String name)
47
48 /**
49  * @effects return id
50 */
51 @DOpt(type=OptType.Observer) @AttrRef("id")
52 public void getId()
53
54 /**
55  * @effects return name
56 */
57 @DOpt(type=OptType.Observer) @AttrRef("name")
58 public String getName()
59
60 /**
61  * @effects
62  *   if id is valid
63  *     return true
64  *   else
65  *     return false
66 */
67 private boolean validateId(int id)
68
69 /**
70  * @effects
71  *   if name is valid
72  *     return true
73  *   else
74  *     return false
75 */
76 private boolean validateName(String name)
77
78 @Override
79 public String toString()
80
```

```

81
82     @Override
83     public boolean equals(Object o)
84
85     /**
86      * @effects
87      *   if this satisfies abstract properties
88      *   return true
89      * else
90      *   return false
91      */
92     public boolean repOK()
93 }
```

6.B IntSet

The complete specification of IntSet.

```

1 import java.util.Vector;
2 import utils.DOpt;
3 import utils.DomainConstraint;
4 import utils.OptType;
5 import utils.collections.Collection;
6 /**
7  * @overview IntSet are mutable, unbounded sets of integers.
8  * @attributes
9  *   elements Set<Integer> Vector<Integer>
10 * @object A typical IntSet object is c={x1,...,xn}, where x1,...,xn are
11 *   elements.
12 * @abstract_properties
13 *   mutable(elements)=true /\ optional(elements)=false /\ 
14 *   elements != {} -> (for all x in elements. x is integer) /\ 
15 *   elements != {} -> (for all x, y in elements. x != y)
16 *
17 * @author dmle
18 */
19 public class IntSet implements Collection {
20     @DomainConstraint(optional = false)
21     private Vector<Integer> elements;
22
23     /**
24      * @effects initialise this to be empty
```

```

25  /*
26  public IntSet()
27
28 /**
29 * @modifies this
30 * @effects
31 * if x already in this
32 *   return false
33 * else
34 *   add x to this, i.e., this_post=this+{x}
35 *   return true
36 */
37 @DOpt(type=OptType.MutatorAdd)
38 public boolean insert(int x)
39
40 /**
41 * @modifies this
42 * @effects
43 * if x is not in this
44 *   return false
45 * else
46 *   remove x from this, i.e. this_post=this-{x}
47 *   return true
48 */
49 @DOpt(type=OptType.MutatorRemove)
50 public boolean remove(int x)
51
52 /**
53 * @effects
54 * if x is in this
55 *   return true
56 * else
57 *   return false
58 */
59 @DOpt(type=OptType.ObserverContains)
60 public boolean isIn(int x)
61
62 /**
63 * @effects
64 * if this is empty
65 *   throw an IllegalStateException
66 * else
67 *   return an arbitrary element of this

```

```
68     */
69     @DOpt(type=OptType.Observer)
70     public int choose() throws IllegalStateException
71
72     /**
73      * @effects return the cardinality of this
74      */
75     @DOpt(type=OptType.ObserverSize)
76     public int size()
77
78     @Override
79     public String toString()
80
81     @Override
82     public boolean equals(Object o)
83
84     /**
85      * @effects
86      * if this satisfies abstract properties
87      * return true
88      * else
89      * return false
90      */
91     public boolean repOK()
92 }
```

Chapter 7

Design Review and Coding

Objectives

- ✓ Appreciate the importance of design review.
- ✓ Be able to use a tool for checking the program design.
- ✓ Understand the general guidelines and a coding procedure of an OOP.
- ✓ Apply the specific coding guidelines for each of the essential operation types (constructor, multator, observer, default and helper).
- ✓ Undertake OOP coding in the Java language.

Before coding can begin, it is necessary to check the design for defects. Any defects that are found need to satisfactorily be resolved. Because the review is performed on the basis of well-defined design rules, it would be useful to develop these rules into a program and use this program to help automate the review tasks. Once the review has been completed, coding can proceed. The general coding guidelines for procedural programs are applicable but should be adjusted to suit the design elements of an OOP.

This chapter first presents a design review approach and a Java program named `OOPChecker`, which is used to automate the review of OOPs written in Java. After that, this chapter describes the coding guidelines and a coding procedure for OOP. The general guidelines are explained first, which are followed by those for each operation type.

7.1 Reviewing the Design

As we saw in the previous chapter, design is a complex process which involves many steps and producing different types of deliverables. It is possible that mistakes occur in one part of the design or in the associations among the different parts. If these mistakes are not detected and corrected early, it will be more costly to fix them once the code has been written.

Design review is thus necessary as this helps identify and fix design errors before implementation commences. In general, the review consists in the following checks:

- check header specification
- check state space specification

- check behaviour space specification

7.1.1 Check Header Specification

The following check list must be observed when reviewing the header specification. The details of each check were discussed in the relevant sections of this chapter.

1. @overview: states what the abstract concept is
2. @attributes: lists the essential attributes of the abstract concept and their data types:
 - (a). formal type is correct
 - (b). concrete type is suitable for the formal type and is supported by the OOPL
3. @object: the definition is based on the attribute(s)
4. @abstract_properties: correct domain rules on the attributes

7.1.2 Check State Space Specification

The following check list must be observed when reviewing the object representation.

1. Instance variables match the attribute definitions
2. @DomainConstraint elements match the corresponding abstract properties

7.1.3 Check Behaviour Space Specification

The following check list must be observed when reviewing the design specification of each operation.

1. performs an essential or a useful task related to the objects
2. the behaviour preserves the abstract properties
3. is annotated with suitable @DOpt and @AttrRef

For example, it is necessary to check that operation IntSet.insert accepts only an integer argument and that it does not lead to a duplicate element being added to the set. Similarly, operation IntSet.remove accepts only an integer argument. In addition, this operation must avoid the possibility of an incorrect element being removed from the set.

As for class Customer, operation Customer.getName needs to correctly return attribute name in the result.

7.2 OOPChecker: A Design Validation Tool

An important benefit of using annotations in program design is that it makes the design rules explicit in the program code. This helps treat the design as a form of code and thus can be checked for structural correctness. Further, this check can be written as a program in OOPLs like Java and C#, which provides a *reflection* feature to browse and analyse the program structure. With this feature, the program elements are expressed as objects and the relationships between elements as object links. The annotation elements, in particular, play two important roles: (1) make explicit the design type of a program element and (2) define design-specific relationships between the program elements. Both types of information can be processed directly in an OOPL.

In this section, we describe a Java tool named `OOPCHECKER`, which automates the design checking described above.

7.2.1 Overview

`OOPCHECKER` is designed to process an extensible set of design rules, which currently include the essential design rules that are represented by the set of annotations discussed in this book. Other design rules can and should be introduced in order to evolve this tool to support better quality program design.

The design rules concern the following types of program elements: class, field (attribute), method and a limited set of code elements in the method body. The tool is executed at compile-time and produces as output the design errors and/or warnings that the developer can then resolve for the design. To improve productivity, the tool is integrated into the Eclipse IDE in the form of a plug-in. A benefit of this is that the tool can leverage the Eclipse's GUI to detect the input program and to display the output directly on a designated GUI component.

Figure 7.1 shows how the `OOPCHECKER` is used in the Eclipse IDE. The tool can be invoked via a tool bar button or a menu item. The input program is selected from the “Project Explorer” tab while the output messages are displayed on the “Problems” tab. With this, the design errors are treated as compile-time errors which must be resolved before the program can be executed.

The tool can be used for checking tutorial exercises or assignment programs. Note that the tool does not validate the correctness of the code, as this requires not only a formal design specification but a logic evaluator tool. This tool basically treats the code and the design specification as two logic programs and checks if the former satisfies

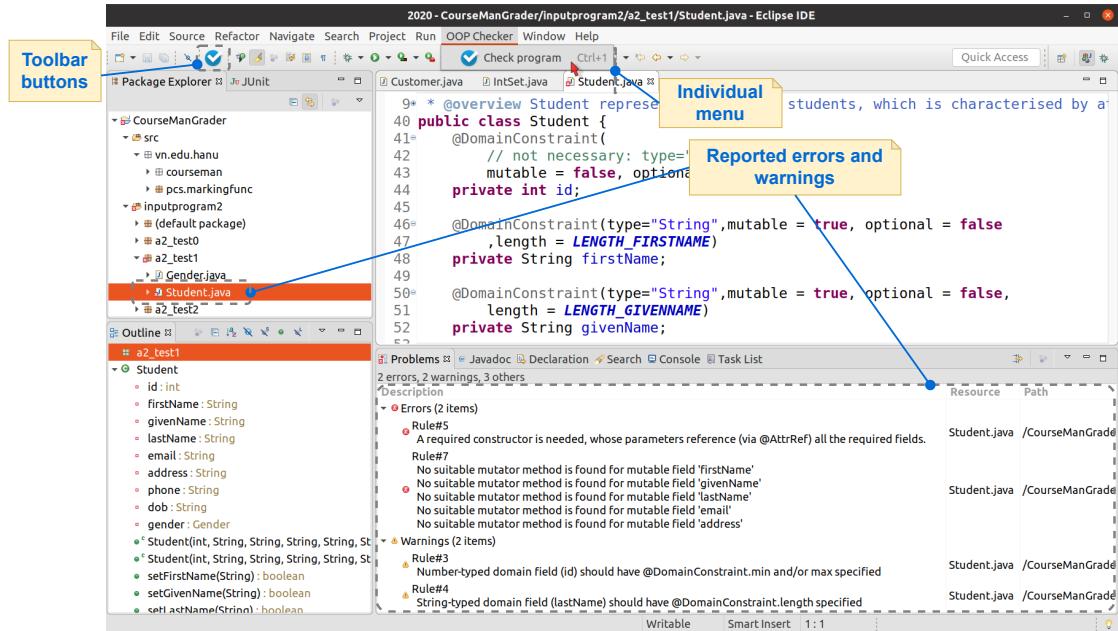


Figure 7.1: OOPCHECKER as an Eclipse plugin.

the later. There is a body of work devoted to the study and design of this type of tool. However, these are outside the scope of this book.

7.2.2 Essential Design Rules

Table 7.1 lists a number of core design rules. All except for the design rule DR_5 are derived directly from the essential domain constraints.

Table 7.1: The essential design rules set that are supported by OOPCHECKER.

Rule	Description
DR_1	State space must be specified
DR_2	Behaviour space must be specified
DR_3	Number-typed domain field should have <code>@DAttr.min</code> and/or <code>max</code> specified
DR_4	String-typed domain field should have <code>@DAttr.length</code> specified
DR_5	A required constructor method must include parameters for all mandatory, non-auto, non-collection-typed fields. Further, each parameter must be attached to an <code>@AttrRef</code> , whose value is set to name of the corresponding field
DR_6	An observer method of a field must be attached to an <code>@AttrRef</code> , whose value is set to the field's name
DR_7	A mutator method of a field must be attached to an <code>@AttrRef</code> , whose value is set to the field's name

Design rule DR_5 states the design rule for the essential constructor. It basically states that if a class has mandatory, non-auto and non-collection-typed fields then the class must have at least one constructor, whose parameters are mapped to those fields

(one parameter is used to initialise the value of one field). For example, `Customer.id` and `name` are mandatory, non-auto and non-collection-typed fields. Thus, class `Customer` needs to have at least one essential constructor method of the form `Customer(id: int, name: String)`. The two parameters `id` and `name` are mapped to (i.e. used to initialise) the fields `Student.id` and `name` (respectively).

7.2.3 Set up and usage

Follow these steps to set up and use OOPCHECKER:

1. Download the `eclipseplugin-oopchecker.zip`¹ file
2. Use the Help/Install New Software menu item of the Eclipse IDE to install the zip file as a plugin
3. Restart the Eclipse IDE
4. To use the plugin:
 1. Select a class or a package
 2. Click the “Check program” button or the menu item in the “OOP Checker” menu

Note that in step 4.1 above, if the user selects a package then all classes in the package are validated. To ease viewing, output messages are reported separately for each class in the package.

7.3 General Implementation Guidelines

Once the design has been checked for structural soundness, it is ready to implemented. This basically involves writing the code for each of the operations. The following general guidelines should be observed. The subsequent sections will discuss some guidelines for the specific operation types.

- Write code to conform to the behaviour description
- Make the most of the built-in operations of the chosen data types, e.g. use `Vector.add` to implement `IntSet.insert` and `Vector.remove` for `IntSet.remove`
- Use the helper operations (e.g. to validate input data)
- Use the keyword `this` to refer to instance variables that have the same names as some local ones

¹<https://github.com/ducml/eopgrader/blob/main/dist/oopchecker/eclipseplugin-oopchecker.zip>

7.4 Constructor

The focus of section is on coding the essential constructor. Coding other constructors would follow similar guidelines. Recall that the general behaviour of the essential constructor is to validate the input arguments and then either initialise the object state or throw an exception. Data validation involves invoking the data validation methods, while throwing an exception follows this syntax: `throw new NotPossibleException (mesg);`. Variable `mesg` represents an error message that contains the constructor name and the argument value. Another point to note is that in order to specialise the error message for each argument, separate exception needs to be throwns.

In the case of collection class (e.g. `IntSet`), the essential constructor simply initialises the object state to be an empty collection.

Example 7.1 Constructor implementation

Listings 7.1 and 7.2 show the Java code of the essential constructors of the two classes `Customer` and `IntSet` (*resp.*). To ease reading, the behaviour descriptions are repeated from the previous chapter. As can be seen from Listing 7.1, constructor `Customer` invokes `validateId` and `validateName` to validate the arguments `custID` and `custName`. If a violation is found then an exception is thrown immediately. The fields `id` and `name` are initialised to the arguments only after the code has passed both validations. The constructor code in Listing 7.2 simply initialises `elements` to an empty `Vector<Integer>`. Note that Java allows a short-hand initialisation syntax of `Vector<>`, which means to takes the type parameter (`Integer`) from the declared type of `elements`.

Listing 7.1: Customer

```

1 /**
2 * @effects
3 *   if custID, name are valid
4 *     initialise this as <custID,name>
5 *   else
6 *     throws NotPossibleException
7 */
8 public Customer(@AttrRef("id") int custID, @AttrRef("name") String name)
9     throws NotPossibleException {
10    if (!validateId(custID)) {
11        throw new NotPossibleException("Customer.init(): invalid id: " + custID);
12    }
13    if (!validateName(name)) {
14        throw new NotPossibleException("Customer.init(): invalid name: " + name);

```

```

14     }
15     id = custID;
16     this.name = name;
17 }
```

Listing 7.2: IntSet

```

1 /**
2 * @effects initialise this to be empty
3 */
4 public IntSet() {
5     elements = new Vector<>();
6 }
```



7.5 Mutator

Similar to constructor, the mutator's general behaviour is to update the object state from some input argument. After validating the argument, it updates the object state (if appropriate) and returns a boolean value indicating the validation result.

Example 7.2

Mutator implementation

Listings 7.3 and 7.4 show the Java code of the mutator operations of the two classes `Customer` and `IntSet` (*resp.*). To ease reading, the behaviour descriptions are repeated from the previous chapter. As can be seen from Listing 7.3, the code invokes method `validateName` to check the input argument `name`. Field `name` is only initialised to the argument if the code has passed this validation.

In Listing 7.4, both `MutatorAdd` and `MutatorRemove` operations invoke the helper operation `getIndex` to retrieve the index of the input element. This index is used to determine the existence of the element and also to remove it from the set.

Listing 7.3: Customer

```

1 /**
2 * @modifies this
3 * @effects
4 *   if name is valid
5 *     set this.name=name
6 *   return true
7 * else
8 *   return false
9 */
10 @DOpt(type=OptType.Mutator) @AttrRef("name")
```

```

11 public boolean setName(String name) {
12     if (validateName(name)) {
13         this.name = name;
14         return true;
15     } else {
16         return false;
17     }
18 }
```

Listing 7.4: IntSet

```

1 /**
2  * @modifies this
3  * @effects
4  *   if x is already in this
5  *     return false
6  *   else
7  *     add x to this, i.e., this_post = this + {x}
8  *     return true
9 */
10 @DOpt(type=OptType.MutatorAdd)
11 public boolean insert(int x) {
12     if (getIndex(x) < 0) {
13         elements.add(x);
14         return true;
15     } else {
16         return false;
17     }
18 }
19
20 /**
21  * @modifies this
22  * @effects
23  *   if x is not in this
24  *     return false
25  *   else
26  *     remove x from this, i.e. this_post = this - {x}
27  *     return true
28 */
29 @DOpt(type=OptType.MutatorRemove)
30 public boolean remove(int x) {
31     int i = getIndex(x);
32     if (i < 0) {
33         return false;
```

```

34     } else {
35         elements.set(i, elements.lastElement());
36         elements.remove(elements.size() - 1);
37         return true;
38     }
39 }
```



7.6 Observer

In general, the observer's code basically returns information obtained from the relevant object state. In some cases, care should be taken not to expose the object state to unexpected modifications by the client program. This would involve transforming the attribute value into a different form before returning it. We will discuss this and other design issues in a later chapter.

Example 7.3 Observer implementation

Listings 7.5 and 7.6 show the Java code of the observer operations of the two classes `Customer` and `IntSet` (*resp.*). The observers' code in Listing 7.5 are basically the same as their behaviours. The observers in Listing 7.6 make use of the helper operations (e.g. `getIndex`) and the relevant Vector's operations to obtain the necessary state information. Note that operation `getElements` does not return `elements` directly, although this is an allowed syntax in Java. Instead, the method transforms `elements` into an array (using the `Collection.toArray` operation) and returns this. The main reason for this transformation is to *completely* shield the attribute `elements` from outside access by the client program. If the operation returned `elements` then the client program has access to the reference to the `Vector` object pointed to by `elements`. The program can modify this `Vector` directly, without using the `IntSet` mutator methods.

Listing 7.5: Customer

```

1 /**
2  * @effects return id
3 */
4 @DOpt(type=OptType.Observer) @AttrRef("id")
5 public int getId() {
6     return id;
7 }
8
9 /**
10 *
```

```

11 * @effects return name
12 */
13 @DOpt(type=OptType.Observer) @AttrRef("name")
14 public String getName() {
15     return name;
16 }

```

Listing 7.6: IntSet

```

1 /**
2  * @effects
3  * if x is in this
4  *   return true
5  * else
6  *   return false
7 */
8 @DOpt(type=OptType.ObserverContains)
9 public boolean isIn(int x) {
10    return (getIndex(x) >= 0);
11 }
12
13 /**
14  * @effects
15  * if this is empty
16  *   throw an IllegalStateException
17  * else
18  *   return an arbitrary element of this
19 */
20 @DOpt(type=OptType.Observer)
21 public int choose() throws IllegalStateException {
22    if (size() == 0)
23       throw new IllegalStateException("IntSet.choose: set is empty");
24
25    return elements.lastElement();
26 }
27
28 /**
29  * @effects return the cardinality of this
30 */
31 @DOpt(type=OptType.ObserverSize)
32 public int size() {
33    return elements.size();
34 }
35

```

```

36 /**
37 * @effects
38 * if this is not empty
39 * return array Integer[] of elements of this
40 * else
41 * return null
42 */
43 @DOpt(type=OptType.Observer)
44 public Integer[] getElements() {
45     if (size() == 0)
46         return null;
47     else
48         return elements.toArray(new Integer[size()]);
49 }

```



7.7 Default Operation

We will demonstrate the coding of three key default operations: `toString`, `equals` and `hashCode`.

7.7.1 Operation `toString`

This operation is to construct a `String` from the relevant attribute values and return this. The format of this string is defined in the `@object` section of the header specification. A simple technique is to use the string concatenation feature of OOPL. However, this is not efficient when there are many string objects that need to be merged. Two alternative techniques can be considered for this: (1) using a mutable string class named `StringBuilder` and (2) using method `String.format`. The first technique is used for `IntSet.toString`, because this involves merging a variable number of string fragments. The second technique is used for `Customer.toString`, because the number of string fragments to be merged is pre-determined and small.

Example 7.4 IntSet

Listing 7.7 shows the code of `IntSet.toString`. The code creates a `StringBuilder` object, loops through the elements and adds each one to this object using the method `append`. Finally, it returns the string by invoking the method `StringBuilder.toString`

Listing 7.7: IntSet

```

1  @Override
2  public String toString() {
3      if (size() == 0)
4          return "IntSet:{}";
5
6      StringBuilder s = new StringBuilder("IntSet:{");
7      s.append(elements.elementAt(0).toString());
8      for (int i = 1; i < size(); i++) {
9          s.append(", ");
10         .append(elements.elementAt(i).toString());
11     }
12     s.append("}");
13     return s.toString();
14 }
```



Example 7.5 Customer

Listing 7.8 shows the code of `Customer.toString`. The code sets up the string format as specified in the class header specification. This format takes two arguments: one numeric (%d) and one string (%s). These arguments are replaced by the values of the two attributes `id` and `name` (*resp.*).

Listing 7.8: Customer

```

1  @Override
2  public String toString() {
3      return String.format("Customer:<%d,%s>", id, name);
4 }
```



7.7.2 Operation equals

Since Java's default behaviour of this operation is to compare objects by their identifiers, the overriding behaviour should compare objects in a different way. A common alternative technique is to compare objects using the object state. If this state contain value elements that are objects, these elements themselves need to be compared as objects. More specifically, one of these two cases is applied to each such element:

- if it is a non-collection object then compare using `equals`
- if it is a collection object then compare the size (obtained from `ObserverSize`) and the elements of the collection

Two other points worth noting about the implementation of `equals` in Java are:

- use keyword `instanceof` to check that argument has the same type
- if argument object is of the same type then can directly access an attribute value of the casted object (i.e. without having to invoke the getter operation)

Example 7.6 Implementation of equals

Listings 7.9 and 7.10 show the code of the operations `Customer.equals` and `IntSet.equals` (*resp.*). Both operations first perform a null-check of the argument object `o` and whether or not it has the expected type. After this, the object is casted into the target type in order to access state information. For operation `Customer.equals` the attribute value `id` is used as the basis for comparison. This attribute has a primitive type, so the comparison is simply the equality operator ‘`==`’.

For operation `IntSet.equals`, however, the object state is a `Vector` object (attribute `elements`). Thus, the comparison is performed using the operation `equals` of the `Vector` class. This operation performs equality based on the collection size and its elements. More specifically, given that the two `Vectors` have the same size, the corresponding pairs of elements in the two collections need to be equal. Each element pair is compared for equality by using the `equals` operation of the element type.

Listing 7.9: Customer

```

1 @Override
2 public boolean equals(Object o) {
3     if ((o == null) || !(o instanceof Customer))
4         return false;
5
6     int yourID = ((Customer) o).id;
7     return yourID == id;
8 }
```

Listing 7.10: IntSet

```

1 @Override
2 public boolean equals(Object o) {
3     if ((o == null) || !(o instanceof IntSet))
4         return false;
5
6     // use Vector.equals to compare elements
7     return elements.equals(((IntSet)o).elements);
8 }
```



7.7.3 Operation hashCode

The discussion here focuses on hash code implementation of non-collection-typed classes (e.g. `Customer`). For these classes, the operation involves directly applying a hash function on the attribute values. The hash code operation of collection-type classes (e.g. `IntSet`) is more complex and would intuitively involve combining the hash codes of the elements. The interested reader should read the Java's API documentations of these two methods `Set.hashCode`² or `List.hashCode`³ for some insights.

Example 7.7 Implementation of hashCode

Listing 7.11 shows the code of `Customer.hashCode`. In this case, the code simply returns `id`, because this is an integer value that is unique across all the objects. In general, if `Customer.id`'s type a non-collection type (e.g. `String`, `Date`, a user-defined type, etc.) then the hash code value should be the one obtained from invoking the operation `hashCode` of the type.

Listing 7.11: Customer

```

1 @Override
2 public int hashCode() {
3     return id;
4 }
```



7.8 Helper

Recall that there are three types of helper operations that need to be implemented: (1) `repOK`, (2) data validation and (3) utility. This section will briefly discuss the implementation of each operation type.

7.8.1 Data validation

Let us start discussing this operation type first, because the logic that it implements is used by the `repOK` operation. The data validation logic depends on the abstract property and the attribute data type. The following general guidelines should be observed:

1. for property optional and object-typed attribute, the null-check should be used.
If attribute type is `String` and `optional=false` then should additionally perform the zero-length check (using `String.length`)

²<https://docs.oracle.com/javase/8/docs/api/java/util/Set.html#hashCode-->

³<https://docs.oracle.com/javase/8/docs/api/java/util/List.html#hashCode-->

2. for property `min` and `max`, the comparison operators ('<', '<=', '>', '>=') should be used
3. for property `length`, the method `String.length` should be used

Example 7.8 Customer

Listing 7.12 shows the code of two data validation operations of the class `Customer`: `validateId` and `validateName`. Operation `validateId` checks the argument `id` against the `MIN_ID` value, while operation `validateName` checks the argument `name` against two properties `optional=false` and `length=LENGTH_NAME`. Note how two checks are used to validate name against the property `optional=false`: the null and zero-length checks.

Listing 7.12: Customer

```

1 /**
2  * @effects
3  *   if id is valid
4  *     return true
5  *   else
6  *     return false
7 */
8 private boolean validateId(int id) {
9   if (id < MIN_ID) {
10    return false;
11  } else {
12    return true;
13  }
14 }
15
16 /**
17  * @effects
18  *   if name is valid
19  *     return true
20  *   else
21  *     return false
22 */
23 private boolean validateName(String name) {
24   return (name != null &&
25         name.length() > 0 &&
26         name.length() <= LENGTH_NAME);
27 }
```



7.8.2 Operation repOK

In principle, the code of `repOK` involves validating the current object state against the abstract properties. If this state consists of object-typed elements then it also involves invoking `repOK` on these elements, and so on. In other words, this would lead to a cascading effect of invoking `repOK` on all the objects that are linked directly and indirectly to the current object.

Another key point about `repOK` is that because most of the validation logic has already been implemented by data validation operations, the `repOK`'s code should reuse the logic by invoking those operations.

Example 7.9 Customer

Listing 7.13 show the code of the operation `Customer.repOK`. The code is simplified by reusing the `validateId` and `validateName` operations to validate the object state against the abstract properties.

Listing 7.13: Customer

```

1 /**
2  * @effects
3  *   if this satisfies abstract properties
4  *   return true
5  * else
6  *   return false
7 */
8 public boolean repOK() {
9     if (!validateId(id) || !validateName(name)) {
10         return false;
11     } else {
12         return true;
13     }
14 }
```



Example 7.10 IntSet

Listing 7.14 shows the code of the operation `IntSet.repOK`. In this case, because no data validation operation against the attribute `elements` is available, the code needs to implement the abstract properties by itself. The three abstract properties of `IntSet` are marked P1, P2 and P3 in the code comments. This helps easily identify the code sections check these properties. Briefly, property P1 involves a `null` check. Property P2 is automatically satisfied because of the use of `Vector<Integer>`. Property P3 is checked by a nested loop that searches for duplicate elements. If a duplicate is found

then the code returns `false` immediately.

Listing 7.14: IntSet

```

1 /**
2  * @effects
3  *   if this satisfies abstract properties
4  *     return true
5  *   else
6  *     return false
7 */
8 public boolean repOK() {
9   // P1: optional(elements) = false
10  if (elements == null)
11    return false;
12
13 // P2: for all x in elements. x is integer
14 //   -> validated by the use of Vector<Integer>
15
16 // P3: for all x, y in elements. x neq y
17 for (int i = 0; i < elements.size(); i++) {
18   Integer x = elements.get(i);
19   for (int j = i + 1; j < elements.size(); j++) {
20     if (elements.get(j).equals(x))
21       return false;
22   }
23 }
24 return true;
25 }
```



7.8.3 Utility

Apart from the general guidelines discussed in Section 7.3, it is difficult to state any concrete coding guidelines for a utility operation without being specific about its behaviour. Listing 7.15 shows the code of the utility operation `IntSet.getIndex`. The code uses a loop over the `elements` to find and return the position of a matching element. An alternative and much simpler code is to take advantage of the method `Vector.indexOf`, which performs exactly the same behaviour. This alternative consists in a single line of code, which is written in the comment ‘`// ALT`’ at the top of the method body.

Listing 7.15: IntSet

```

1 /**
2  * @effects
3  *   if x is in this
4  *     return the index where x appears
5  *   else
6  *     return -1
7 */
8 private int getIndex(int x) {
9   // ALT: return elements.indexOf(x);
10  for (int i = 0; i < elements.size(); i++) {
11    if (x == elements.get(i))
12      return i;
13  }
14  return -1;
15 }
```

7.9 Implementing the main method

After a class has been implemented, it is time to write a client program that uses that class. In general, this client program would consist of several objects, potentially from different classes, interacting with each other through operation invocation. Within the scope of this chapter, however, we will consider client programs, whose behaviours are simply to create objects of the implemented class and invoke some operations on them. To simplify the discussion, we implement the program behaviour in its `main` method.

We consider two main programs: one program, named `CRM`, is to work with `Customer` and the other, named `Integers`, is to work with `IntSet`. We conclude this section with a program that demonstrates the use of wrapper classes.

7.9.1 A Basic Customer Relationship Manager (CRM)

`CRM` is a simple program for `Customer` class that creates some objects of this class with valid and invalid data. It also invokes some operations on the objects to update or display their states. The program, which is shown in Listing 7.16, consists of the following key elements:

- A static variable, named `idCounter`, is used to keep track of the number of customer objects that are created. This variable is used to initialise an automatically-generated customer id
- method `createCustomer`: creates a new `Customer` object from the input data

- method `main`: initialises a CRM instance and invokes its methods to perform the intended behaviour

Listing 7.16: Program CRM

```

1 import utils.NotPossibleException;
2 /**
3  * @overview A program that creates Customer objects and displays their
4  * details
5  */
6 public class CRM {
7     private static int idCounter = 1;
8     /**
9      * @effects
10     * if successfully creates a new Customer <id,name> (id is auto-
11     * incremented)
12     * return the object
13     * else
14     * throws NotPossibleException
15     */
16     public Customer createCustomer(String name) throws NotPossibleException {
17         int id = idCounter;
18         Customer c = new Customer(id, name);
19         idCounter++;
20         return c;
21     }
22     /**
23      * @effects create some valid and invalid customer objects and
24      * display their details
25      */
26     public static void main(String[] args) {
27         CRM bp = new CRM();
28         String[] names = {"Nguyen Van A", null, "Tran Van Tuan"};
29         Customer c;
30         for (String name: names) {
31             try {
32                 c = bp.createCustomer(name);
33                 System.out.println("Created customer: " + c);
34             } catch (NotPossibleException e) {
35                 System.err.println("Could not create a customer with name: " + name);
36                 e.printStackTrace();
37             }
38         }

```

```

39 }
40 } // end CRM

```

7.9.2 Integers

Integers is a simple program for IntSet class that creates an object of this class and invokes some operations on this object to update or display its elements. The program code is shown in Listing 7.17. The code uses the method Arrays.toString to conveniently obtain a string representation of an array's elements.

Listing 7.17: Program Integers

```

1 import java.util.Arrays;
2 /**
3 * @overview A program that creates a set of integers and displays its
4 * elements
5 * @author dmle
6 */
7 public class Integers {
8 /**
9 * @effects Creates a set of integers and displays its description
10 */
11 public static void main(String[] args) {
12     int a[] = { 1, 1, 2, 2, 3, 5, 6, 8 };
13     System.out.println("a: " + Arrays.toString(a));
14     IntSet s = new IntSet();
15     for (int i = 0; i < a.length; i++) {
16         s.insert(a[i]);
17     }
18     System.out.println(s);
19 }

```

7.9.3 Working with Wrapper Classes

Wrapper classes were discussed in a previous chapter as suitable concrete data types for attributes. Program Wrappers, shown in Listing 7.18, demonstrates how to work with the wrapper class Integer. Similar programs can be constructed for other wrapper classes. The program extends a simpler version that was used in a previous chapter. In addition to the auto-boxing and auto-unboxing features, it also includes code for the parsing and value operations.

Listing 7.18: Program Wrappers

```
1 /**
2  * @overview A program that creates and manipulate objects of wrapper classes
3  *
4  public class Wrappers {
5   /**
6    * @effects initialise some Integer objects and performs the key operations
7    * on them.
8   */
9   public static void main(String[] args) {
10    // create using constructor
11    Integer i = new Integer(10); /* i = 10 */
12    // create object using parse operation
13    i = Integer.parseInt("10"); /* i = 10 */
14    // create object using auto-boxing
15    i = 5;                      /* i = 5 */
16    // unboxing in expression
17    int e = i + 10;              /* e = 15 */
18    // convert back to primitive using intValue
19    int p = i.intValue();        /* p = 15 */
20    // convert using unboxing
21    p = i;                      /* p = 5 */
22 }
```



Chapter 7 Exercise

The exercises in this chapter continue from those in Chapter 6 with two main objectives: (1) review the design and (2) implement the design in Java. Note that some exercises were reproduced from [20].

1. **(Greeting Conversation)** Review the design of and implement the `GreetingConversation` program of [Exercise 6.1](#):
 - (a). Review the design of and implement class `Person`.
 - (b). Review the design of and implement class `MobilePhone`.
 - (c). Create an program class named `GreetingConversation`, whose `main` method performs the following basic object manipulation tasks:
 - create a `MobilePhone` object.
 - create a `Person` object.
 - check that the objects are valid and if so display information about them, else display suitable error messages.
2. **(Greeting Conversation v1.1–1.3)** Review the design of and implement the three improvements to `GreetingConversation` discussed in [Exercise 6.2](#)-[Exercise 6.4](#). Update method `GreetingConversation.main` to make use of the improved design.
3. Review the design of and implement class `EvenIntSet` of [Exercise 6.5](#). Create an program class named `EvenIntegers`, whose `main` method performs the basic object manipulation of `EvenIntSet`.
4. Review the design of and implement class `StringIntMap` [Exercise 6.6](#). Create an program class named `MixedMaps`, whose `main` method performs the basic object manipulation of `StringIntMap`.
5. Review the design of and implement class `IntQueue` [Exercise 6.7](#). Create an program class named `NumQueue`, whose `main` method performs the basic object manipulation of `IntQueue`.
6. Review the design of and implement class `Rat` [Exercise 6.8](#). Create an program class named `Rationals`, whose `main` method performs the basic object manipulation of `Rat`.

7.A Customer

The complete Java code of Customer.

```

1 import utils.AttrRef;
2 import utils.DOpt;
3 import utils.DomainConstraint;
4 import utils.NotPossibleException;
5 import utils.OptType;
6 /**
7 * @overview Customers are people or organisations with which we have
8 * relationships.
9 * @attributes
10 *   id Integer
11 *   name String
12 * @object A typical Customer is c=<d,n>, where id(d), name(n).
13 * @abstract_properties
14 *   mutable(id)=false /\ optional(id)=false /\ min(id)=1 /\
15 *   mutable(name)=true /\ optional(name)=false /\ length(name)=50
16 * @author dmle
17 */
18 public class Customer {
19     @DomainConstraint(type = "Integer", mutable = false,
20         optional = false, min = MIN_ID)
21     private int id;
22
23     @DomainConstraint(type = "String", optional = false, length = LENGTH_NAME)
24     private String name;
25
26     // constants
27     private static final int MIN_ID = 1;
28     private static final int LENGTH_NAME = 50;
29
30     // constructor methods
31     /**
32      * @effects <pre>
33      *   if custID, name are valid
34      *   initialise this as <custID,name>
35      * else
36      *   throws NotPossibleException
37      */
38     //{@DOpt(type=OptType.Constructor)
39     public Customer(@AttrRef("id") int custID, @AttrRef("name") String name)

```

```

39     throws NotPossibleException {
40     // if custID, name are valid
41     if (!validateId(custID)) {
42         throw new NotPossibleException("Customer.init: Invalid customer id: " +
43             custID);
44     }
45     if (!validateName(name)) {
46         throw new NotPossibleException("Customer.init: Invalid customer name: " +
47             name);
48     }
49     // initialise this as <custID,name>
50     id = custID;
51     this.name = name;
52 }
53
54 /**
55 * @effects <pre>
56 *   if name is valid
57 *     set this.name=name
58 *   return true
59 * else
60 *   return false</pre>
61 */
62 @DOpt(type=OptType.Mutator) @AttrRef("names")
63 public boolean setName(String name) {
64     if (validateName(name)) {
65         this.name = name;
66         return true;
67     } else {
68         return false;
69     }
70 }
71
72 /**
73 * @effects return <tt>id</tt>
74 */
75 @DOpt(type=OptType.Observer) @AttrRef("id")
76 public int getId() {
77     return id;
78 }
79

```

```

80  /**
81   * @effects return <tt>name</tt>
82   */
83  @DOpt(type=OptType.Observer)
84  @AttrRef("name")
85  public String getName() {
86      return name;
87  }
88
89  /**
90  * @effects <pre>
91 *   if id is valid
92 *     return true
93 *   else
94 *     return false
95 * </pre>
96 */
97  private boolean validateId(int id) {
98      return id >= MIN_ID;
99  }
100
101 /**
102 * @effects <pre>
103 *   if name is valid
104 *     return true
105 *   else
106 *     return false</pre>
107 */
108 private boolean validateName(String name) {
109     return (name != null &&
110         name.length() > 0 &&
111         name.length() <= LENGTH_NAME);
112 }
113
114 @Override
115 public String toString() {
116     return String.format("Customer:<%d,%s>", id, name);
117 }
118
119 @Override
120 public boolean equals(Object o) {
121     if (o == null || !(o instanceof Customer))
122         return false;

```

```

123
124     int yourID = ((Customer) o).id;
125     return yourID == id;
126 }
127
128 @Override
129 public int hashCode() {
130     return id;
131 }
132
133 /**
134 * @effects <pre>
135 *   if this satisfies abstract properties
136 *   return true
137 * else
138 *   return false</pre>
139 */
140 public boolean repOK() {
141     return validateId(id) && validateName(name);
142 }
143 }
```

7.B IntSet

The complete Java code of IntSet.

Listing 7.19: IntSet

```

1 import java.util.Vector;
2 import utils.DOpt;
3 import utils.DomainConstraint;
4 import utils.OptType;
5 import utils.collections.Collection;
6 /**
7 * @overview IntSet are mutable, unbounded sets of integers.
8 * @attributes
9 *   elements Set<Integer> Vector<Integer>
10 * @object A typical IntSet object is c={x1,...,xn}, where x1,...,xn are
11 *   elements.
12 * @abstract_properties
13 *   optional(elements) = false /\
14 *   for all x in elements. x is integer /\
15 *   for all x, y in elements. x neq y
```

```

16 * @author dmle
17 */
18 public class IntSet implements Collection {
19     @DomainConstraint(optional = false)
20     private Vector<Integer> elements;
21
22     // constructor methods
23     /**
24      * @effects initialise <tt>this</tt> to be empty
25      */
26     public IntSet() {
27         elements = new Vector<>();
28     }
29
30     /**
31      * @modifies <tt>this</tt>
32      * @effects <pre>
33      *   if x is already in this
34      *   return false
35      * else
36      *   add x to this, i.e., this_post = this + {x}
37      *   return true
38      * </pre>
39      */
40     @DOpt(type=OptType.MutatorAdd)
41     public boolean insert(int x) {
42         if (getIndex(x) < 0) {
43             elements.add(x); // auto-boxing
44             return true;
45         } else {
46             return false;
47         }
48     }
49
50     /**
51      * @modifies <tt>this</tt>
52      * @effects <pre>
53      *   if x is not in this
54      *   return false
55      * else
56      *   remove x from this, i.e. this_post = this - {x}
57      *   return true
58      * </pre>

```

```

59  /*
60  @DOpt(type=OptType.MutatorRemove)
61  public boolean remove(int x) {
62      int i = getIndex(x);
63      if (i < 0) {
64          return false;
65      } else {
66          elements.set(i, elements.lastElement());
67          elements.remove(elements.size() - 1);
68          return true;
69      }
70  }
71
72 /**
73 * @effects <pre>
74 *   if x is in this
75 *   return true
76 * else
77 *   return false</pre>
78 */
79 @DOpt(type=OptType.ObserverContains)
80 public boolean isIn(int x) {
81     return (getIndex(x) >= 0);
82 }
83
84
85 /**
86 * @effects return the cardinality of <tt>this</tt>
87 */
88 @DOpt(type=OptType.ObserverSize)
89 public int size() {
90     return elements.size();
91 }
92
93 /**
94 * @effects
95 *   if this is not empty
96 *   return array Integer[] of elements of this
97 * else
98 *   return null
99 */
100 @DOpt(type=OptType.Observer)
101 public Integer[] getElements() {

```

```

102     if (size() == 0)
103         return null;
104     else
105         return elements.toArray(new Integer[size()]);
106     }
107
108 /**
109 * @effects <pre>
110 *   if this is empty
111 *     throw an IllegalStateException
112 *   else
113 *     return an arbitrary element of this</pre>
114 */
115 public int choose() throws IllegalStateException {
116     if (size() == 0)
117         throw new IllegalStateException("IntSet.choose: set is empty");
118     return elements.lastElement();
119 }
120
121 /**
122 * @effects <pre>
123 *   if x is in this
124 *     return the index where x appears
125 *   else
126 *     return -1</pre>
127 */
128 private int getIndex(int x) {
129     // ALT: return elements.indexOf(x);
130     for (int i = 0; i < elements.size(); i++) {
131         if (x == elements.get(i))
132             return i;
133     }
134     return -1;
135 }
136
137 @Override
138 public String toString() {
139     if (size() == 0)
140         return "IntSet:{}";
141
142     StringBuilder s = new StringBuilder("IntSet:{");
143     s.append(elements.elementAt(0).toString());
144     for (int i = 1; i < size(); i++) {

```

```

145     s.append(" , ")
146     .append(elements.elementAt(i).toString());
147   }
148   s.append("}");
149   return s.toString();
150 }
151
152 @Override
153 public boolean equals(Object o) {
154   if (o == null && !(o instanceof IntSet))
155     return false;
156
157   // use Vector.equals to compare elements
158   return elements.equals(((IntSet)o).elements);
159 }
160
161 /**
162 * @effects <pre>
163 *   if this satisfies abstract properties
164 *   return true
165 * else
166 *   return false</pre>
167 */
168 public boolean repOK() {
169   // P1: optional(elements) = false
170   if (elements == null)
171     return false;
172
173   // P2: for all x in elements. x is integer
174   //      -> validated by the use of Vector<Integer>
175
176   // P3: for all x, y in elements. x neq y
177   for (int i = 0; i < elements.size(); i++) {
178     Integer x = elements.get(i);
179     for (int j = i + 1; j < elements.size(); j++) {
180       if (elements.get(j).equals(x))
181         return false;
182     }
183   }
184   return true;
185 }
186 }
```

Chapter 8

Design Issues

Objectives

- ✓ Understand the four basic class design issues.
- ✓ Explain an object's lifecycle, object initialisation and reference, stack and heap memory and how object is passed in a method call.
- ✓ Be able to create private constructors of a class.
- ✓ Be able to clone an object.
- ✓ Be able to design an inner class when needed.
- ✓ Be able to create a generic class.

Having discussed the class design fundamentals, let us discuss in this chapter the complete set of class design issues. First, we review the basic class design issues in the context of the design approach presented in the previous chapter. Second, we discuss a number of additional issues concerning objects. Third, we explain private constructors and when they should be designed in a class. Fourth, we discuss the default method for cloning objects and how to design this method. Last but not least, we present the design of inner class and generic class. Some of the design features presented in this chapter will be used in a later chapter to define design patterns and abstract data types.

8.1 Basic Design Issues

In this section, we discuss four basic class design issues that are frequently studied in the OOP literature.

8.1.1 Information Hiding

As discussed in an earlier chapter, **information hiding** means to hide the internal design details of one class from other classes. The benefit of this is to prevent internal design changes to a class from affecting the code that uses objects of the class. Information hiding is achieved by:

- using the access modifier **private** for attributes, and
- preventing the object representation (the **rep**) from being exposed by protecting the

object-typed, mutable attributes from either being set directly in or being returned from public methods

The first technique was already discussed in the previous chapter. The second technique will be discussed in the context of encapsulation.

8.1.2 Encapsulation

An orthogonal issue to information hiding is encapsulation, which is to design suitable public operations to allow objects of other classes to interact with its objects. Which operations to use and how to design them are the concern of encapsulation. The class design method discussed in the previous chapter helps answer these questions at the fundamental level. There is one additional subtle issue that needs to be addressed, which concerns public operations that operate on mutable attributes. Let us explain what this issue is and how to resolve it.

Protecting object-typed, mutable attributes

These are attributes whose declared types are object types (e.g. `Customer`, `Vector`, etc.) and have the property `mutable=true`. Additional care should be taken when designing the public operations that operate on these attributes, so as not to unexpectedly leak internal object references to the using code. Such a leak occurs when the object reference of an attribute is known by the using code. Let us illustrate with an example design of the class `IntSet` shown in Listing 8.1. For brevity, the design annotations are omitted from the listing. Even though attribute `elements` is declared `private`, it is still exposed to the using code because the constructor and the getter operation leaks its object reference. By initialising `elements = els`, the attribute has the same object reference as the one created by the using code (`els`). Operation `getElements`, on the other hand, returns the object reference to the using code.

Listing 8.1: An IntSet design that exposes the rep

```

1  public class IntSet {
2      private Vector elements;
3      public IntSet(Vector els) {
4          elements = els;
5      }
6      public Vector getElements() {
7          return elements;
8      }
9  }
```

Thus, both innocent-looking operations in this example potentially breaks the class encapsulation by allowing the using code to directly modify the value of attribute `IntSet.elements`. To avoid object reference leakages such as this, extra care should be taken to:

- not directly assign attribute to an argument of a public operation, and
- not return the attribute value directly from a public operation

Let us discuss how to implement each of these rules in the context of collection-typed classes. We will use `IntSet` to demonstrate.

How to assign the elements of a collection? To assign elements of a collection to a collection-typed collection without exposing the rep, it is necessary to copy the content of the collection over. Listing 8.2 demonstrates how this is achieved for `IntSet.elements`. It contains two constructors: one takes a `Vector` object as argument and the other takes an `Object[]` array as argument. In both cases, the code involves initialising `elements` as a separate (empty) `Vector` and looping through the input collection to copy each of its elements over to `elements`.

Listing 8.2: `IntSet` improvement (1)

```

1  public class IntSet {
2      private Vector elements;
3      public IntSet(Vector els) {
4          elements = new Vector();
5          for (Object o : els) {
6              elements.add(o);
7          }
8      }
9
10     public IntSet(Object[] els) {
11         elements = new Vector();
12         for (Object o : els) {
13             elements.add(o);
14         }
15     }
16 }
```

How to provide access to the elements of a collection? In the other direction, to return a collection's elements without exposing the collection itself, it is necessary to create a new collection to hold the elements and return this collection. Listing 8.3 illustrates this solution for the getter operation `IntSet.getElements`. The code returns

an `Object []` array that contains a copy of elements in the collection. Any modifications to this array outside `IntSet` do not affect the content of `IntSet.elements`. Note that a simpler code than using the loop to copy the elements is to use the method `toArray` of the class `Vector`.

Listing 8.3: `IntSet` improvement (2)

```

1 public class IntSet {
2     private Vector elements;
3     public Object[] getElements() {
4         if (elements.size() == 0) {
5             return null;
6         } else {
7             Object[] els = new Object[elements.size()];
8             for (int i = 0; i < elements.size(); i++) {
9                 els[i] = elements.get(i);
10            }
11            return els;
12        }
13    }
14 }
```

8.1.3 Separation of Concerns

As discussed in a previous chapter, this refers to the separation between the class design specification and its code. The design specification defines the behaviour, while the code states how this behaviour is realised in a specific programming language. A main benefit of separating the specification of the behaviour from implementation is that it allows users to understand an operation without having to read the code. This separation also helps the programmer to focus on providing the correct implementation of a behaviour.

8.1.4 Operation Overloading

The code example in Listing 8.2 contains two constructors of the class `IntSet`, which differ in the parameter list. This is an example of constructor overloading – a special case of a more general feature called operation overloading. In theory, **operation overloading** (a.k.a **method overloading** in OOPL) helps enhance the benefit of specification by abstraction by generalising the same behaviour for different input types. For instance, the operation `add` can be generalised for both integral and floating-point data types. In Java, overloading operations have the same name but different

parameter lists. The return types of these operations may or may not be the same. In fact, many operations of the built-in classes of Java are overloaded. `Operation System.out.println` is a great example of this. It is overloaded to support all kinds of primitive and referenced data types.

Because the overloading operations have the same behaviour, one operation may invoke others if needed. This is illustrated in the following example.

Example 8.1 Overloading `toString`

Listing 8.4 shows two overloading `toString` operations of the class `Customer`. The second operation, which overrides the default operation `Object.toString`, invokes the first one with a boolean argument `true`. The first operation either produces a full string representation of the object (containing the values of all the attributes), if the argument is `true`, or produces a partial string representation containing only the values of `id` and `name`.

Listing 8.4: Overloading `Customer.toString`

```

1 /**
2 * @effects
3 * if full = true
4 *   return Customer:<id,name,dob>
5 * else
6 *   return Customer:<id,name>
7 */
8 public String toString(boolean full) {
9     StringBuilder sb = new StringBuilder();
10    sb.append("Customer:<").
11    append(id).append(",") .
12    append(name);
13    if (full) {
14        sb.append(",").
15        append(dob);
16    }
17    sb.append(">");
18
19    return sb.toString();
20 }
21
22 /**
23 * @effects
24 * return Customer:<id,name>
25 */
26 @Override

```

```

27 public String toString() {
28     return toString(true);
29 }

```

□

8.2 Example: Customer

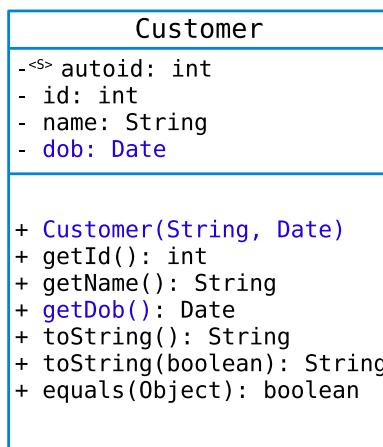


Figure 8.1: An extended design of class *Customer*.

To illustrate the remaining design issues in this chapter, we introduce an extended design of the class *Customer* that we used in a previous chapter. Figure 8.1 shows the design of this class. This design includes one additional attribute named *dob*. This attribute captures the date of birth and is typed *Date*. This class is provided in the *java.util* package. Another design feature the introduction of a static (class) variable named *autoId*. This variable is initialised to 1 and is automatically incremented by 1 for each new *Customer* object. Its value is used to initialise the attribute *Customer.id*.

8.3 Object Life Cycle

To work effectively with OOPs requires understanding how objects are created and managed by the target OOPL platform. Figure 8.2 shows a state diagram that provides a high-level view of the object life cycle. The life cycle starts when an object is created and ends when the object is removed from the run-time memory. First, an object is created by the keyword *new* and invoking a constructor operation. This is illustrated in the figure by the arrow labelled “*new*”. This arrow transitions the object from the initial state to the *created* state. Next, as the object is used in programs, its state may be changed by one of the mutators. The two arrows labelled “*mutate*” demonstrate this, showing

the transitions from the state created to the state changed and reflexively from the state changed to itself. After the object has served its purpose and is no longer needed (i.e. no program variables are pointing to the object), it transitions to the unused state, waiting to be removed by the run time environment. The two arrows in the figure are labelled “nullify”. They show the transitions from the states created and changed to unused, respectively.

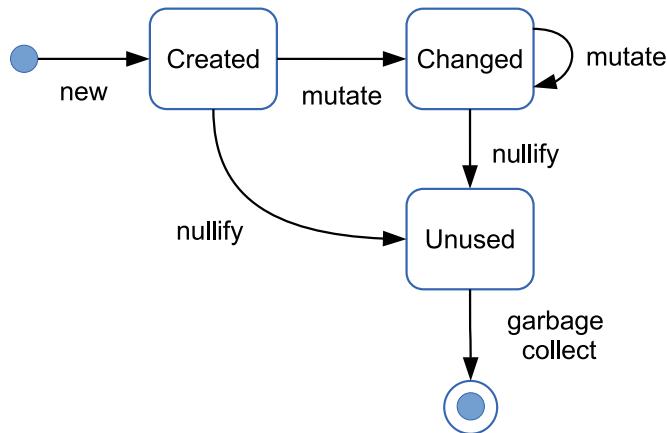


Figure 8.2: The object life cycle showing the high-level states and the transitions between them.

Unused objects may or may not be removed immediately from memory. For safety reasons, both Java and C# automatically manage the removal of unused objects. The run time environment maintains a separate schedule (named **garbage collection**) that is responsible for removing unused objects from memory. The arrow labelled “garbage collect” in the figure demonstrates garbage collection, showing the transition from the state unused to the final (terminated) state.

8.3.1 Object Initialisation

At the most basic level, an object is initialised at creation time as follows:

- the attributes are initialised to their default values
- the attributes are assigned to input arguments as specified by the constructor

Table 8.1 lists the default values for the various primitive and object data types.

Example 8.2 (Object initialisation)

Listing 8.5 demonstrates the default attribute values when an object is created. Class `ObjectInit` has eight attributes, the first seven of which have primitive types. The last attribute has the general type `Object`. Procedure `main` creates an object of `ObjectInit` and displays the value of each attribute. Note that to simplify the design, we obtain the attribute values directly via the object rather than via the observers. This is possible in Java because procedure `main` is a procedure of the same class `ObjectInit`.

Table 8.1: Default attribute values.

Data types	Default (initialisation) values
byte	0
char	'\u0000'
short	0
int	0
long	0l
float	0f
double	0d
object types (e.g. String, arrays, ...)	null

Listing 8.5: ObjectInit

```

1 public class ObjectInit {
2     private byte b; private char c; private short s;
3     private int i; private long l; private float f;
4     private double d; private Object o;
5     public static void main(String[] args) {
6         ObjectInit o = new ObjectInit();
7         System.out.println("byte: " + o.b);
8         System.out.printf("char: %c (%d)%n", o.c, (int) o.c);
9         System.out.println("short: " + o.s);
10        System.out.println("object: " + o.o);
11    }
12 }
```

□

8.3.2 Object Reference

An object reference is the memory address of the location of the object stored in memory. This address is computed by the OOPL's run-time when an object is created and is assigned to an object-typed variable. When the variable is passed around in the program (via procedure invocations), copies of this object reference are made available to the using code. Consequently, these code can access and modify the object by invoking its operations.

Figure 8.3 illustrates object reference with a `Customer` object. When this object is created with the `new` keyword, the object reference (*not* the object) is assigned to the variable `cust`. This reference is illustrated in the figure by an arrow connecting the variable to the object. Note that the figure actually shows two other object references, which point to two objects of the types `String` and `Date`. These objects were created and used as input for initialising the attributes `name` and `dob` (*resp.*) of the `Customer`

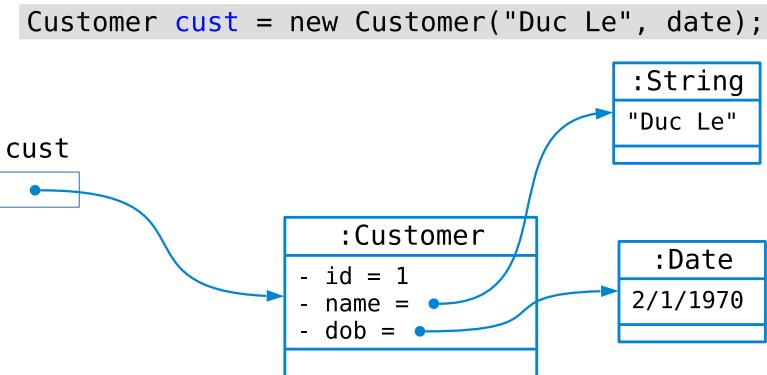


Figure 8.3: Example object reference created for a `Customer` object.

object.

In both Java and C#, a program creates objects and works with their references during execution. However, as discussed earlier, the object life time is managed separately by the run-time environment (through garbage collection). A program can declare an object to be no longer in use, but it can not control when the object is actually removed from memory. Let us then discuss the memory arrangement for the objects and their references.

8.3.3 Stack and Heap

These are two main memory types that are made available by the OOPL platform to running programs. A technical discussion of these memories were given in the chapter about virtual machine. This section will explain how they are used by programs to manage data values and objects. Although the discussion here focuses on the Java VM, a similar understanding also exists for the C# counterpart.

Recall that **stack memory** stores data associated with an operation invocation. The data include input arguments and local variables. On the other hand, **heap memory** is a shared memory area that holds objects. Although an object is typically created during the execution of an operation, the object itself is not placed in the stack. The object is placed in the heap, while its reference (which is assigned to a local variable) is kept in the stack. Conceptually, the variables in the stack, the objects and the heap and their references form a directed graph. To help visualise this graph and the impacts of the state changes that occur during program execution, we introduce **stack and heap diagram**. This is basically a directed graph that consists of two sets of nodes: one set is located in the stack and the other is located in the heap. Let us illustrate this diagram with two examples.

Example 8.3 Stack and heap (1)

Listing 8.3 shows a class `StackAndHeap`, which demonstrates the stack and heap diagram. It implements a code segment given in [20]. To understand the effect of the code on stack and heap, let us divide it into two segments. Figure 8.4 shows two snapshots of the stack-and-heap diagram, which maps out the contents of stack and heap for each code segment. The first segment consists in the lines 3-8, which initialise two primitive-typed variables `i`, `j` and four object-typed variables `a`, `b`, `s` and `t`. The second segment consists in lines 9-11, which are variable assignments involving these variables.

```

1 public class StackAndHeap {
2     public void op() {
3         int i = 6;
4         int j; // uninitialized
5         int[] a = {1,3,5,7,9}; // creates a 5-element array
6         int[] b = new int[3];
7         String s = "hello"; // creates a new string
8         String t = null;
9         j = i;
10        b = a;
11        t = s;
12    }
13 }
```

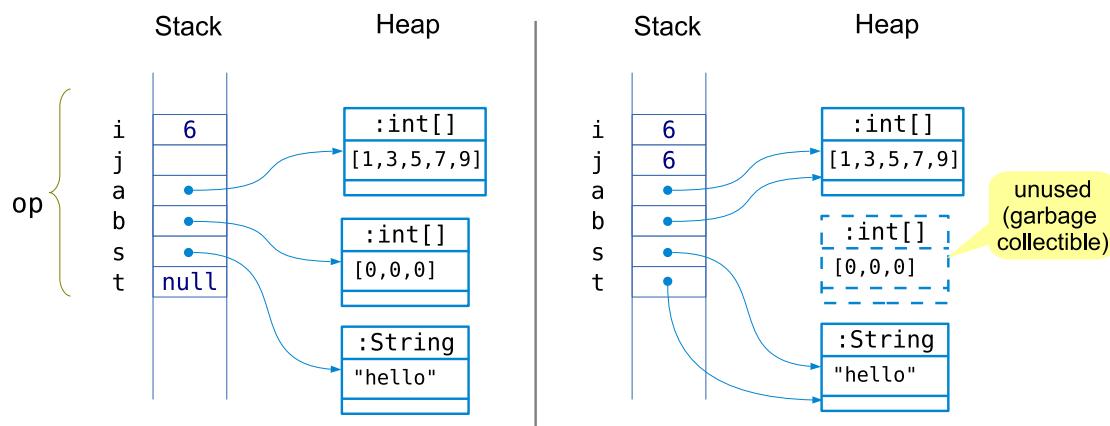


Figure 8.4: Two snapshots of the stack and heap diagram.

The left-hand-side snapshot in Figure 8.4 shows the stack of the operation `op` and the heap containing three objects. The stack lists six variables and their values. Variable `i`'s value is 6, while `j` is uninitialised (illustrated as an empty box). Variable `t` is null. The values of `a`, `b`, and `s` are references that point to three objects in the heap. These references are illustrated by three arrows that connect the variables to the objects. Note that `b`'s elements are initialised to the default values of 0.

The right-hand-side snapshot in the figure shows updates to the variable values

after the second segment is executed. Specifically, variable `b` now contains a copy of the reference to the same array object as variable `a`. Both variables point to the same object. Similarly, `t` now contains a copy of the reference to the same `String` object as `s`.

The array object `[0,0,0]` that `b` initially pointed to is now unused and is subject to be garbage collected. □

Example 8.4 Stack and heap (2)

Let us consider another example of stack and heap diagram which more clearly shows the graph-based nature of the objects and their references. The diagram in Figure 8.5 represents a partial snapshot of stack and heap after executing the `CustomerAppSimple` program shown in Listing 8.6. This code creates a `Customer` object, which takes a `String` and an `Date` object as input.

To ease reading, variable `cal` is omitted from the figure. The first variable is the input argument `args`, which contains a reference to an empty `String[]` object in the heap. This object is created automatically by the run-time when the method `main` is invoked. The second variable is `is` a reference to a `Customer` object. This object consists in three values: `id`, `name` and `dob`. The value of `id` is 1, because this is the first customer object that is created. The value of `name` is a reference to the `String` object “Duc Le” in the heap, while the value of `dob` is a reference to the `Date` object `2/1/1970` in the heap.

As far as coding is concerned, the program uses the built-in class `Calendar` in order to generate a desired `Date`¹ object representing the date of birth ‘`2/1/1970`’ of the customer. Line 2 is to obtain the system-specific object of the `Calendar` class and to assign it to a variable `cal`. This object is created by the `Calendar` class itself, using system’s specific calendar settings. The class procedure `Calendar.getInstance` is used for this purpose. Line 3 then invokes the operation `set` on the object `cal`, passing in three arguments: `1970`, `0` and `2`. The first argument is the value of the year, the second is the month index (0 to 11) and the third is the value of the day. These have the effect of pointing the current date of the calendar to `2/1/1970`. When the customer object is created at line 4, we invoke the `getTime` operation on the object `cal` to obtain the desired `Date` object.

Listing 8.6: CustomerAppSimple

```

1 public static void main(String[] args) {
2     Calendar cal = Calendar.getInstance();
3     cal.set(1970,0,2);

```

¹Java 8 or above uses a new date class called `LocalDate` (<https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html>).

```

4     Customer cust = new Customer("Duc Le", cal.getTime());
5 }

```

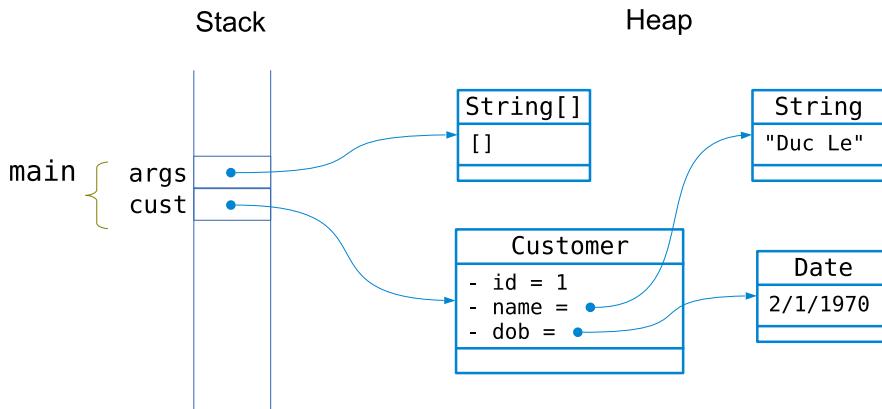


Figure 8.5: The stack and heap diagram of `CustomerAppSimple`.

□

8.3.4 Passing An Object as Argument

As discussed above, an object-typed variable only holds the object reference and not the object itself. Consequently, when a variable is passed as input in a procedure invocation, a copy of its value is assigned to an argument. The object itself stays in the heap. With the object reference value, however, the procedure's code has complete access to the object and, thus, can manipulate the object by invoking its operations. Any changes to the object state that are performed by the procedure also affect other procedures that depend on that state. This is a key point to remember when working with procedure invocations that involve objects!

Example 8.5 Procedural invocation involving objects.

Let us illustrate using a program, named `CustomerApp`, whose code is shown in Listing 8.7. In addition to demonstrating procedure invocation with objects, this program shows how to construct and display a formatted report from objects. Procedure `main` basically contains the code explained earlier in Example 8.4 plus a call to a procedure named `report`. This invocation passes a copy of the object reference of the variable `cust` as argument. Procedure `report` then uses this reference to obtain information about the `Customer` object and displays a formatted report on the standard output. Lines 28-33, in particular, show how to use the procedure `String.format` to generate formatted string fragments that are components of a larger `String`. The format pattern is exactly the same as that used by `System.out.printf`. The only difference here is that `String.format` produces a formatted string as output, without displaying it on the

standard output. The string fragments are combined with the help of a `StringBuilder` object.

Listing 8.7: Program CustomerApp

```

1 import java.util.Calendar;
2 import java.util.Date;
3 /**
4 * @effects
5 * Represents a simple application that uses Customer.
6 * @author dmle
7 */
8 public class CustomerApp {
9 /**
10 * @requires c neq null
11 * @effects
12 * generate and display to the terminal console a formatted report about c
13 */
14 private static void report(Customer c) {
15     StringBuilder rept = new StringBuilder();
16     Date d = c.getDob();
17     Calendar cal = Calendar.getInstance();
18     cal.setTime(d);
19     String dstr = cal.get(Calendar.DATE) + "/" +
20         (cal.get(Calendar.MONTH)+1) + "/" + cal.get(Calendar.YEAR);
21
22     rept.append("      CUSTOMER REPORT      ").append("\n");
23     final String idf = "%3s";
24     final String namef = "%10s";
25     final String dobf = "%15s";
26     // report fields
27     rept.append(String.format(idf,c.getId())).
28         append(String.format(namef,c.getName())).
29         append(String.format(dobf,"date of birth")).append("\n");
29 // report values
31     rept.append(String.format(idf,c.getId())).
32         append(String.format(namef,c.getName())).
33         append(String.format(dobf,dstr));
34     System.out.println(rept.toString());
35 }
36
37 /**
38 * @effects
39 * create a Customer object and display a formatted report.
40 */

```

```

41  public static void main(String[] args) {
42      Calendar cal = Calendar.getInstance();
43      cal.set(1970,0,2);
44      Date date = cal.getTime();
45      Customer cust = new Customer("Duc Le", date);
46      // report
47      report(cust);
48  }
49 }
```

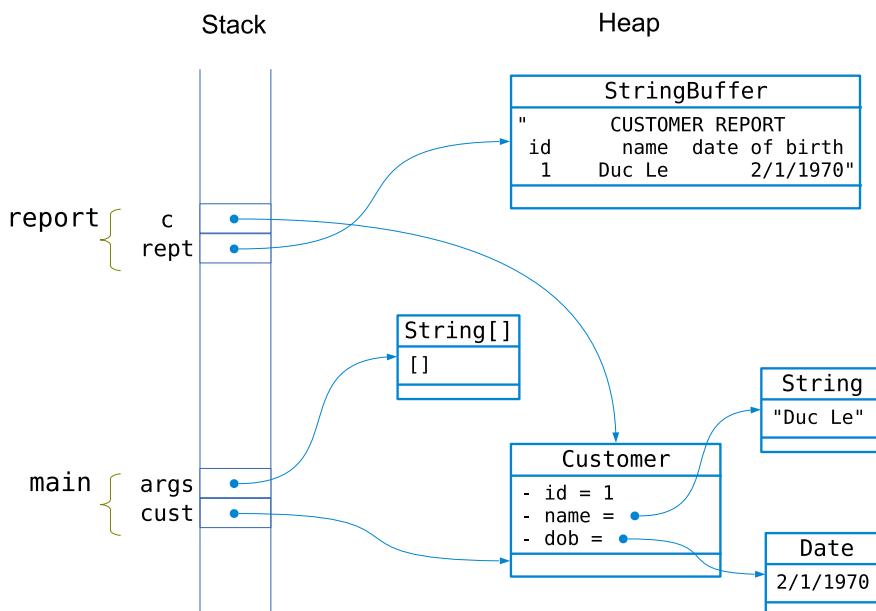


Figure 8.6: A partial stack and heap diagram of CustomerApp.

Figure 8.6 shows a partial stack and heap diagram for the procedures `main` and `report`. Similar to Figure 8.5 of Example 8.4, the argument `args` of procedure `main` points to an empty `String []` object, while variable `cust` points to a `Customer` object. A difference shown in Figure 8.6, however, is the stack variables concerning the invocation of procedure `report`. These include the argument `c`, which contains a copy of the object reference kept in `cust`, and variable `rept`. This second variable refers to a `StringBuilder` object, whose content is the output of the report that is displayed to the user. □

8.4 Private Constructor

A class typically defines a public constructor to enable programs to create its objects. When the needs arise, however, a class can define private constructors as well. A **private constructor** is a constructor with access modifier `private` and is, thus, accessible only

to the code within the class. Private constructor is typically used to provide a restricted form of object creation. This section will discuss a common design scenario for private constructor, named *hidden initialisation*. Other scenarios will be discussed later in the chapter about design patterns.

Hidden Initialisation

A class should have a private constructor if this constructor is used only by code within the class. Certain operations, e.g. object cloning (discussed later in Section 8.5), require a behaviour that creates default objects. However, the abstract property specification may prevent such a behaviour from being made public. In this case, the only plausible solution for the behaviour is a private default constructor.

Example 8.6 Customer's private default constructor

Listing 8.8 shows the definition of a private default constructor of class `Customer`. It is basically a default constructor that is defined with the access modifier `private`.

Listing 8.8: `Customer`

```

1 /**
2 * An empty constructor used to clone (i.e. copy) objects
3 * @effects
4 * initialise this as Customer:<0,null,null>
5 */
6 private Customer() {
7     // do nothing
8 }
```



8.5 Object Cloning

The class design chapter discussed three default operations that are provided by the base class `Object`. Another default operation is named `clone`. The purpose of this operation is to make copies of an object. This behaviour is useful for situations where the exact or similar copies of an object are frequently created in a program. Operation `clone` provides an exact copy of an object. Client program can either use this object as is or customise its state to serve the purpose.

A special note about operation `Object.clone`² is that it can not be directly invoked (by the client program) and, thus, needs to be reimplemented (i.e. overridden) for this

²<https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#clone-->

purpose. According to the design specification of `Object.clone`, an implementation of this method should, but is not required to, produce a behaviour that has the following properties:

```
(1-different) o.clone() != o
(2-typing) o.clone().getClass() == o.getClass()
(3>equals) o.clone().equals(o) == true (equality)
```

The design specification of `Object.clone` suggests an implementation technique that involves using an interface named `Cloneable`. This section introduces another, simpler technique that does not require this interface.

Example 8.7 Customer's cloning

Listing 8.9 shows the design and implementation of the operation `Customer.clone`. The operation header is tagged `@Override` and has the return type `Customer`. The behaviour states that the operation returns a deep copy of the current object, which is an object that contains copies of the values of primitive-typed as well as object-typed attributes. To realise this, the code first creates an empty `Customer` object (using the private constructor of Example 8.6). It then copies the value of each attribute and assign the copies to the attributes of the new object. The value of attribute `name` is used to create a new `String` object, while the `Date` value of attribute `dob` is cloned.

Listing 8.9: `Customer.clone`

```

1  /**
2   * @effects
3   *   return a deep copy of this, i.e. Customer<i,n,d>, where
4   *   i = id, n = new String(name), d = dob.clone()
5   */
6  @Override
7  public Customer clone() {
8      // different object but same class
9      Customer c = new Customer();
10     // same content with deep copy
11     // copy value
12     c.id = id;
13     // clone string value
14     c.name = new String(name);
15     // clone dob
16     c.dob = (Date) dob.clone();
17     return c;
18 }
```





Question Implement operation `IntSet.clone` that performs a deep clone of `IntSet` objects.

8.6 Inner Class

A class can be defined as a member of another class, making it an integral part of that class. The member class is called an **inner class**, while the enclosing class is called the **outer class**. Inner class is generally only useful when it models a dependent type, whose semantics is defined as part of another class. This means that the inner class can not exist on its own. Its existence depends on that of the outer class. An example of this is a class named `Address`, which exists only to model the addresses of the `Customers`. In this case, `Address` depends on `Customer` for its existence and, thus, should be defined as an inner class.

An inner class is defined using the usual class definition syntax, except that the whole definition is placed inside the outer class. A special design feature of inner class is that it has full access to members of the outer class. Conversely, the outer class also has full access to members of the inner one.

Example 8.8 Inner class

Class `Customer2` in Figure 8.7 shows an extension of the `Customer` class in Figure 8.1. The extension includes an inner class named `Address` and an attribute called `address` that is declared with this inner class. Class `Address` encapsulates the common address attributes, including `number`, `street`, `district`, `city`, and `country`.

Listing 8.10 shows the Java code of class `Customer2`. Note that class `Address` is declared `public` so that `Address` objects can be manipulated by the client program. The attribute values are initialised via the constructor. The class defines several getters for the attributes and an operation `toString`. The program class that runs this example is the class `chap8.CustomerApp2` in the attached source code.

Listing 8.10: Inner class `Customer2.Address`

```

1 public class Customer2 {
2     // code omitted
3     private Address address;
4     // code omitted
5     /**
6      * @overview Represents a customer address
7      * @author dmle
8      */

```

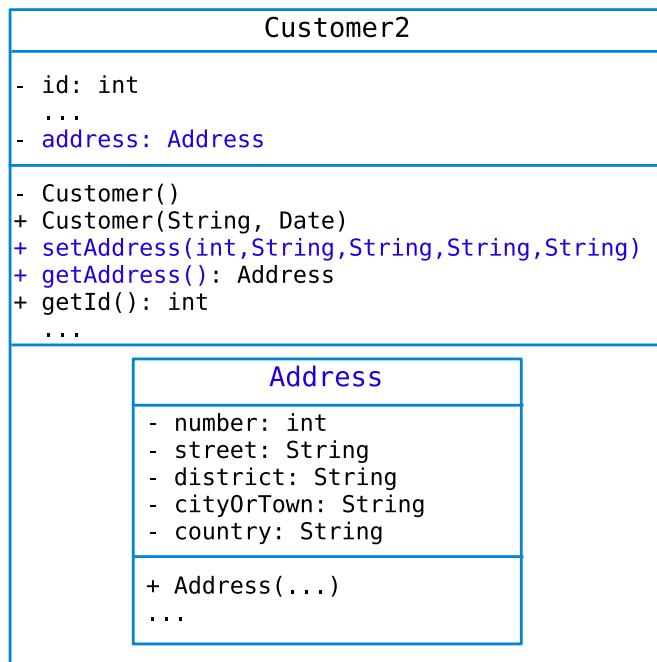


Figure 8.7: The design of inner class `Customer2.Address`.

```

9  public class Address {
10     private int number;
11     private String street;
12     private String district;
13     private String cityOrTown;
14     private String country;
15
16     public Address(int number, String street, String district, String
17                   cityOrTown, String country) {
18         this.number = number;
19         this.street = street;
20         this.district = district;
21         this.cityOrTown = cityOrTown;
22         this.country = country;
23     }
24
25     public int getNumber() {
26         return number;
27     }
28
29     public String getStreet() {
30         return street;
31     }
32
33     public String getDistrict() {

```

```

33     return district;
34 }
35
36     public String getCityOrTown() {
37         return cityOrTown;
38     }
39
40     public String getCountry() {
41         return country;
42     }
43     @Override
44     public String toString() {
45         StringBuilder sb = new StringBuilder("Address:<");
46         sb.append(number)
47             .append(",").append(street)
48             .append(",").append(district)
49             .append(",").append(cityOrTown)
50             .append(",").append(country)
51             .append(">");
52         return sb.toString();
53     }
54 } // end Address
55 } // end Customer

```



8.7 Generic Class

In OOPL, **generic class** is classified under a more general feature called *generic type*. A **generic type** (which can be a class or interface) is a type that is declared with one or more type parameters. This means that the behaviour defined by the generic type can be applied to all the types captured by the type parameters. The main benefit is to enhance code reuse while ensuring type safety.

A generic class, in particular, is used in the same way as a normal class, except for the specification of the actual types in place of the type parameters. This is also called class parameterisation, the result of which is a **parameterised class**.

The general design guidelines for a generic class is as follows:

1. Define the class header with one or more type parameters. These parameters are written within a pair of angle brackets; e.g. `ClassC<T, ...>`
2. Specify attributes with type parameters where necessary

3. Specify the operations with type parameter(s) in the formal parameters and/or in the return type

Example 8.9 Generic Set

Let us demonstrate generic class using a simplified version of a well-known, built-in Java type called `Set`³. It makes sense to apply generics to this class because its behaviour is applicable to different element types, namely integers, doubles, strings, customers, etc. The idea is to define a generic class `Set<T>`, where the type parameter `T` represents the element type. Figure 8.8 shows the design of `Set<T>`. This is basically adapted from the class `IntSet` (defined in an earlier chapter) by introducing the type parameter `T` to the class definition and replacing all references to the `int` data type in the object representation and operation specification by `T`. Listing 8.11 gives a partial code of `Set<T>`, whose aim is to highlight the key design changes. The complete code of the class is given in `chap8.generic.Set` of the attached source code.

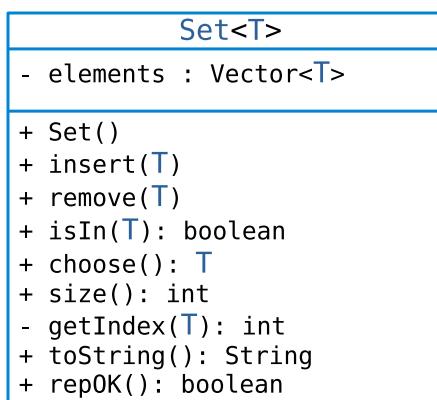


Figure 8.8: The design of generic class `Set<T>`.

Listing 8.11: Generic class `Set<T>`

```

1  /** (omitted) */
2  public class Set<T> implements Collection<T> {
3      @DomainConstraint(type = "Vector", optional = false)
4      private Vector<T> elements;
5
6      /** (omitted) */
7      @DOpt(type=OptType.MutatorAdd)
8      public void insert(T x) {
9          if (getIndex(x) < 0)
10              elements.add(x);
11      }
12
13      @DOpt(type=OptType.MutatorRemove)
  
```

³<https://docs.oracle.com/javase/8/docs/api/java/util/Set.html>

```

14  public void remove(T x) {
15      int i = getIndex(x);
16      if (i < 0)
17          return;
18      elements.set(i, elements.lastElement());
19      elements.remove(elements.size() - 1);
20  }
21  // (omitted)
22 }
```



~~~~ Chapter 8 Exercise ~~~~

The exercises in this chapter help practise applying the solutions of the class design issues to improve a class's design.

1. It is said that declaring the attributes of a class as `private` is enough to protect these attributes from outside access. Is this true? Briefly explain why or why not. Use a code example to illustrate.
2. Below is the design specification and code of a class named `SClass`. The header specification is correct. The class specification, however, has some errors. Identify and fix them.

```

1 import java.util.Vector;
2 import userlib.DomainConstraint;
3 /**
4  * @overview SClass is student class, which is a group of students that
5  * studies together throughout a university degree.
6  *
7  * @attributes
8  *   id      Integer  int
9  *   name    String
10 *  students Student[]  Vector<Student>
11 *
12 *  @object
13 *  A typical SClass is <i,n,s>, where id(i), name(n), students(s). For
14 *  example, <1,1c10,[]> is an SClass representing the student class
15 *  whose id is 1, whose name is 1c10, and whose student list is empty
16 *
17 *
18 *  @abstract_properties
19 *  *  mutable(id)=true /\ optional(id)=false /\ min(id)=1 /\
20 *  *  mutable(name)=true /\ optional(name)=false /\
21 *
```

```

17 *      length(name)=20 /\ 
18 *  mutable(students)=true /\ optional(students)=false
19 */
20 public class SClass {
21     @DomainConstraint(type="Integer",mutable=true,optional=false,min=1)
22     public int id;
23     @DomainConstraint(type="String",mutable=true,optional=false,length
24         =20)
25     public String name;
26     @DomainConstraint(type="Vector",mutable=true,optional=false)
27     public Vector students;
28     /**
29     * @effects
30     * if name and students are valid
31     * initialise this as <0,name,students>
32     * else
33     *   print error message "Invalid input"
34     */
35     public SClass(String name, Vector students) {
36         this.name = name;
37         this.students = students;
38     }
39     /**
40     * @effects
41     * return id
42     */
43     public int getId() {
44         return id;
45     }
46
47     /**
48     * @effects
49     * return name
50     */
51     public String getName() {
52         return name;
53     }
54
55     /**
56     * @effects
57     * return students
58     */

```

```

59     public Vector getStudents() {
60         return students;
61     }
62
63     @Override
64     public String toString() {
65         return "SClass:<" + id + ", " + name + ", " + students + ">";
66     }
67 }
```

- 3.** Below is the code of a program that performs some computation on integer arrays.

Answer the following questions about this program:

- (a). What are the program states in the stack and heap memories when the code is run?
 - (b). What is the console output after running the program?
-

```

1 public class Program2 {
2     public static void main(String[] args) {
3         int[] x = { 1, 3, 5 };
4         int[] y = { 1, 2, 3, 4 };
5
6         int[] z = y;
7         int j;
8         for (int i = 0; i < x.length; i++) {
9             for (j = 1; j < z.length; j++) {
10                 z[j] = z[j] + z[j - 1];
11             }
12             x[i] = x[i] * z[j-1];
13
14             System.out.println(x[i]);
15         }
16
17         for (int i = 0; i < y.length; i++)
18             System.out.println(y[i]);
19     }
20 }
```

- 4.** Design and code at least one overloading constructor operation for class Person of the GreetingConversation program of [Exercise 6.1](#). Justify your design.
- 5.** Design and code at least one overloading constructor operation for class MobilePhone of the GreetingConversation program of [Exercise 6.1](#). Justify your design.
- 6.** Design and code at least one overloading constructor operation for class StringIntMap

- of [Exercise 6.6](#). Justify your design.
7. Design and code at least one overloading operation for the `MutatorAdd` and `MutatorRemove` operations of the class `IntSet` of Chapter 6. Justify your design.
 8. Change the design and code of the class `StringIntMap` of [Exercise 6.6](#) to use an inner class named `Entry` to represent an entry in the map.

Chapter 9

Introduction to Design Patterns

Objectives

- ✓ Understand design pattern and the pattern definition format.
- ✓ Apply a set of basic class design patterns.
- ✓ Derived attribute pattern: to construct a class with a derived attribute.
- ✓ Singleton pattern: to construct a singleton class.
- ✓ Factory pattern: to construct a factory class.
- ✓ Recursive class pattern: to construct a recursive class.

If design is considered to be the activity of defining the solution to a software problem then questions can be asked in terms of how to make this activity performed more efficiently. More specifically, is there a way to design the solution so that it can easily be reused to define the solutions for other similar problems? If so, is it beneficial to survey the frequently occurring design problems and define the solutions for them in a reusable manner? Answering these two questions arguably leads to the notion of *design pattern* – a well-documented solution to a frequently-occurring problem. There have been many books written about design patterns, prescribing different patterns for various types of software problems. In fact, design pattern has become such an integral part of the software engineering's body of knowledge that software engineering professionals are expected to learn and make them part of their tools of trade.

This chapter gives a practical introduction to design pattern in the context of the OOP design method that has been presented so far in this book. It explains what design pattern is, its structured description format and a set of four fundamental class design patterns. The first two patterns concern the design of two types of utility classes, called singleton and factory. The third pattern concerns the design of derived attribute. The fourth pattern concerns recursive class. Some of these patterns will be reused in a later chapter to define the design solutions of complex abstract data types.

9.1 What Is Design Pattern?

The study of design pattern has its root in architectural design, where this term was used to describe building design patterns. Christopher Alexander [2] precisely puts it:

“each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”

Adapting this term to OOP design, Gamma et al. [10] characterises **design pattern** as a construct that “... names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design.” Gamma et al. also describe a pattern definition format and presents a catalog of generic design patterns. The definition format, in particular, provides a uniform and structural means for documenting the patterns. This in turn makes it easy for designers to study and evaluate the suitability of a pattern to a problem at hand. Gamma et al. uses a pattern definition format as the documentation language for the patterns in the catalog. Designers need to understand this language in order to read the catalog. The catalog is organised along these three main class design dimensions: creational, structural and behavioural.

Within the scope of this book, four design patterns are chosen for presentation in this chapter. Two of these patterns are taken from the creational pattern type in [10]. The other two patterns are patterns that can be considered to be of the structural type. They are useful enough to be included here for presentation. The first pattern concerns derived attribute, while the second pattern concerns recursive class.

9.2 Pattern Definition Format

For brevity, this book adapts the abstract pattern definition format, which is presented in Chapter 1 of [10]. In this format, a design pattern is defined in terms of the following five elements:

1. **name:** a unique pattern name, which encapsulates what the pattern is about. A good pattern name is important because it helps enrich the design vocabulary and enables the pattern adopters to work with their designs in an abstract form.
2. **problem:** describes the aim and motivation of the problem that gives rise to the pattern. The specific properties of the problem need to be stated, so that adopters can determine if the pattern matches with what they need.
3. **solution:** an abstract design model that describes a solution to the stated problem. It is abstract in the sense that the model elements are given generic names.
4. **example:** describes an application example of the pattern. This example includes the design and code (written in Java).
5. **consequences:** state the advantages (or benefits) and disadvantages (or costs)

of applying the pattern. This helps adopters evaluate the pattern against other alternative design solutions.

9.3 Class Design Patterns

This chapter presents four basic design patterns that describe solutions to four common problems concerning object creation and structure. According to Gamma et al. [10], creational patterns that provide an abstraction over the object creation process, thereby insulating the client program from their complexity. On the other hand, structural patterns concern with constructing components from classes and objects.

The main reason for presenting the four patterns in this chapter is that their designs concern only the design elements of a single class. They help enrich the design vocabulary of class designers. The four selected patterns are:

1. **singleton**: a creational pattern (discussed in Section 9.4)
2. **factory method**: a creational pattern (discussed in Section 9.5)
3. **derived attribute**: a structural pattern (discussed in Section 9.6)
4. **recursive class**: a structural pattern (discussed in Section 9.7)

The first two patterns are defined in Gamma et al. but without a Java-specific solution. The other two patterns are the new contributions of this book.

9.4 Singleton

Singleton pattern concerns classes that have only one object. Explicitly designing these classes to limit the number of valid objects helps improve performance and, in some cases, protect the domain integrity.

9.4.1 Problem

Let's discuss the intent and motivation for singleton.

Intent

To restrict the number of valid objects of a class to one and to provide a shared reference to this object.

Motivation

In practice, software requirements typically prescribe rules on the number of objects of a class (called the *subject*) that can be associated to an object of another class. A special case of these rules is when a subject class can have at most one and the same object in any associations with other objects. In this case, the subject class has at most one object created in the software. This type of class, called **singleton**, is often found representing the key subsystems of a software system. For example, the accounting subsystem [10] of a software system has one common object that serves accounting requests from all other subsystems.

The key design questions facing singleton are:

1. How to design a singleton class such that at most one object can be created?
2. How to make this object easily accessible by other software components?

A simple design solution, which is often found in traditional non-object-oriented software, is to use a global (public) variable for the object. This variable is initialised once (possibly at loading time) and is then globally available for use by any components that need it. However, this solution breaks the encapsulation principle by making the variable vulnerable to modification by the using components. A better solution, therefore, is needed that follows the standard object oriented design principles.

9.4.2 Solution

To answer design question 1 requires enforcing that the singleton class has complete control over object creation. A simple solution for this is to hide the class constructor (using the private constructor feature) behind a public operation. This operation is the single service point for all object creation requests and that its behaviour must always produce the same object of the class. Because this operation creates an object, it needs to be defined as **static**. To keep the object created by this operation, the class needs to have a **static** variable, which is assigned to the object the first time it is created.

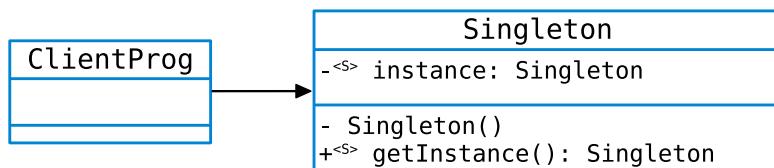


Figure 9.1: Design diagram of singleton pattern.

To answer question 2 requires making the above operation the point of reference for the object. This is achieved by simply returning the object reference when it is invoked.

Figure 9.1 shows the design of the singleton class and how it is used by ClientProg. The design elements of the class are summarised as follows:

- a static variable (named `instance`) that points to the object.
- a single private constructor that initialise the object.
- a public static operation (named `getInstance`) that creates the object (when invoked the first time) and returns the object.

9.4.3 Example

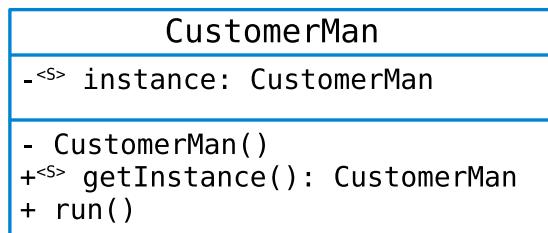


Figure 9.2: Singleton class CustomerMan.

Listing 9.1 shows the design specification and code of a class named `CustomerMan`. This class represents a customer management component of software. It references the `Customer` class that was defined in a previous chapter. It makes sense to restrict the number of `CustomerMan` objects to one, because only one such object is ever needed. The UML design of `CustomerMan` is summarised in Figure 9.2.

Listing 9.1: Singleton class CustomerMan

```

1 import java.util.Calendar;
2 import java.util.Date;
3 /**
4 * @overview
5 * A simple class uses Customer and has only one instance.
6 * @author dmle
7 */
8 public class CustomerMan {
9     /** a single object of this class */
10    private static CustomerMan instance;
11    /**
12     * @effects
13     * initialise this to be an empty object
14     */
15    private CustomerMan() {
16        //
17    }
18
  
```

```

19  /**
20  * @effects
21  * if instance = null
22  * initialise it
23  * return instance
24  */
25 public static CustomerMan getInstance() {
26     if (instance == null) {
27         instance = new CustomerMan();
28     }
29     return instance;
30 }
31
32 /**
33 * @effects
34 * create a Customer and display some info about it
35 */
36 public void run() {
37     Calendar cal = Calendar.getInstance();
38     cal.set(1970,0,2);
39     Date date = cal.getTime();
40     Customer cust = new Customer("Duc Le", date);
41     System.out.println(cust);
42     System.out.println(cust.toString());
43     System.out.println("age: " + cust.getAge());
44 }
45 }

```

As shown in the listing, the class variable `instance` is declared to be of type `CustomerMan`. It is declared as `static` so that it belongs to the class rather than to the objects. The private constructor has an empty parameter list and an empty body because the `CustomerMan` instance needs only be initialised with a default state. If there were attributes then the constructor would take the required parameters and initialise the object state with them. The class operation `getInstance` initialises `instance` to an object of `CustomerMan`, if this has not been created, and returns this object. This guarantees at most one object of the class is created. The object operation `run` demonstrates the execution of `CustomerMan`, which, for illustration purpose, simply creates a `Customer` object and displays some information about it. The following code demonstrates a simple client program that uses `CustomerMan`:

```

CustomerMan app = CustomerMan.getInstance();
app.run();

```

9.4.4 Consequences

The singleton pattern brings the following benefits to class design:

- **simple enforcement of a single object** without the need to use a separate object management class. This helps simplify the design and use.
- **conserves memory** by preventing unnecessary objects from being created. Creating several identical objects of the singleton class is not wrong but doing so is a waste of run-time memory.
- **extensible** to support a predetermined number of instances [10]. This can be achieved by adding to the class a limit on the number of instances that can be created and updating the `getInstance` operation to use this limit.

However, the pattern has the following disadvantage:

- **life cycle dependency**: the object is recorded as part of the class and thus its life cycle is attached to that of the class. The object is not garbage collected unless the class is unloaded.

9.5 Factory Method

As the name implies, factory method refers to an operation that serves the key function of an “object factory” – to produce objects. This pattern concerns the design of this type of operation.

9.5.1 Problem

Let’s discuss the intent and motivation for factory method.

Intent

To define a method that is responsible for creating objects of a class, whose actual type may not be known at compile time.

Motivation

When an instance variable is declared in a program its type, called the *declared type* [20], may represent a group of compatible types. Objects of these types can all be assigned to the variable, making them the **actual types** (of the variable). The declared type is referred to as the **abstract type**. To illustrate let us consider the simple client

program example below. In this program, `Object` is the declared type of variable `v`, while `String` and `StringBuilder` are the actual types of the `v`.

```

1 Object v; // abstract type: Object
2 v = new String("hello"); // actual type: String
3 v = new StringBuilder("hello"); // actual type: StringBuilder

```

Now, consider a realistic situation where only one of the two objects (`String` or `StringBuilder`) is needed by the program and that which object gets created depends on a run-time condition that is not within the control of the program. Liskov and Guttag [20] gives two examples of these conditions. The first example is to remove the unnecessary dependency between the client program and the actual type. The client program only knows the abstract type and so when the actual types are changed (or by adding a new one), it is not affected. The second example is the platform type on which the program is executed. This occurs when multiple versions of the program are developed for different platforms.

The presence of a run-time condition means that the actual type is not known at compile-time. This calls for a design solution that separates the object creation logic from the client program that uses the objects. To this end, two basic design questions can be raised:

1. What abstraction should we use to represent the object creation logic, such that it hides knowledge about the actual object type from the client program?
2. How does this abstraction easily support multiple object types?

9.5.2 Solution

The answer to design question 1 is an operation, called **factory method**, that is defined in a separate class and that its return type is the abstract type. Although it is technically possible to define a factory method in each of the actual type class, doing so will make the client program dependent on the actual types.

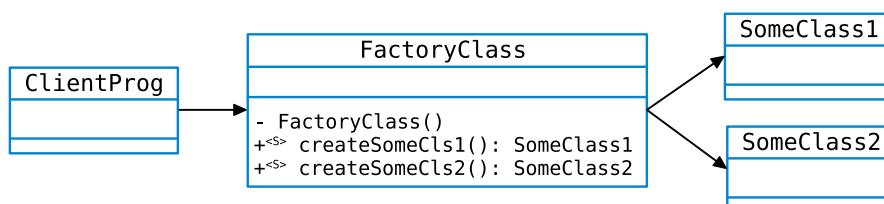


Figure 9.3: Design diagram of factory pattern.

The factory method can be an object or class operation. Object operation is suitable when the factory class needs to be instantiated for use in the client program. On the other

hand, class operation is used when the factory method can be invoked directly through the factory class. If this is the case then it may also be useful to prevent factory class's objects from being created. This is achieved by making the factory class constructor private.

Both Liskov and Guttag [20] and Gammar et al. [10] suggest several design alternatives to answer design question 2. A simple alternative, which is discussed in [20] and is suitable for our study scope of this book, is to define several factory methods in a **factory class**, one for each object type.

Figure 9.3 shows the abstract design of the factory class and how it references some object types (SomeClass1 and SomeClass2) and how it is used by ClientProg . FactoryClass has a private constructor and two class operations acting as factory methods for objects of someClass1 and SomeClass2.

9.5.3 Example

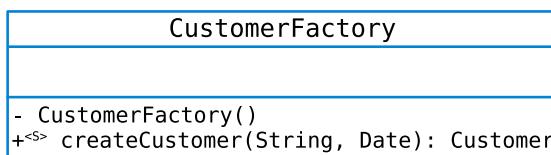


Figure 9.4: Factory class CustomerFactory.

Listing 9.2 is a factory class called CustomerFactory. The UML design of this class is shown in Figure 9.4. The factory method, named createCustomer, creates a Customer object from two input arguments and returns it. CustomerFactory's constructor is declared as private, because no objects of this class are needed.

Listing 9.2: Factory class CustomerFactory

```

1 /**
2  * @overview
3  * A factory class to create Customer objects
4  * @author dmle
5 */
6 public class CustomerFactory {
7 /**
8  * No objects are needed
9 */
10 private CustomerFactory() {
11     //
12 }
13
14 /**
  
```

```

15  * @effects
16  * create and return new Customer:<name,dob>
17  */
18 public static Customer createCustomer(String name, Date dob) {
19     Customer cust = new Customer(name, dob);
20     return cust;
21 }
22 }
```

The following code demonstrates a simple client program that uses `CustomerFactory`:

```

Calendar cal = Calendar.getInstance();
cal.set(1970,0,2);
Date dob = cal.getTime();
Customer cust = CustomerFactory.createCustomer("Duc Le", dob);
System.out.println("Created: " + cust);
```

9.5.4 Consequences

The factory method pattern brings the following benefits to class design:

- ***separation of concerns***, which leads to modular design and better code reuse.
Client program can use an extensible set of object types, without having any prior knowledge about them.
- ***centralised object creation rule enforcement***, which is useful in cases where the object creation logic is complex, involving the coordination of many other dependent objects. Placing this logic in one operation helps not only hide the complexity but ensure that the creation logic is correctly observed.

However, the pattern has the following disadvantage(s):

- ***added layer of indirection*** would affect run-time performance. Instead of invoking the constructor directly, the client program needs to call the factory method.

9.6 Derived Attribute

A **derived attribute** of a class is an attribute whose value is automatically derived (or computed) from other attributes of the same class. This pattern concerns the design of this attribute.

9.6.1 Problem

Let's discuss the intent and motivation for derived attribute.

Intent

To define an attribute whose value in each object state is automatically derived from one or more attributes of the same class.

9.6.1.1 Motivation

In principle, derived attributes capture information about some parts of the object state (called the *observed parts*). For example, two pieces of information about `Customer` objects that are of frequent interest to the client program are surname and age. Surname is derived from attribute `Customer.name`, while age is derived from attribute `Customer.dob`. Surname would be used to perform sorting of the objects, while the age would be used in different parts of the program to determine the suitability of a `Customer` to a service.

To provide these information, a familiar solution is to define suitable observer operations, whose behaviours compute and return the desired values. In the above example, for instance, class `Customer` would be updated with two observers named `getSurname()` and `getAge()`, whose behaviours extract the surname and compute the age, respectively.

The issue here is that there are cases where the computation of the information is nontrivial (and is thus time consuming) and/or the information is frequently used in the program and the observed parts rarely change. In these cases, it is beneficial to design the observer operations in order to avoid computing the information every time an operation is invoked. Ideally, the computation is only performed once in the first invocation and is repeated when the observed parts have been updated. To this end, two key design questions can be raised:

1. How to record the computed information and save it for subsequent use?
2. How to update the recorded information when the observed parts are changed?

9.6.2 Solution

The natural answer to design question 1 is to use an instance variable, whose value is assigned to the computed information. This variable is called **derived attribute**. This can not be a static variable, because the variable value is set differently for each object.

It can not be a local variable of the observer operation either, because this variable is not retained between subsequent invocations of the operation. Note, in particular, that the derived attribute must be immutable (i.e. property `DomainConstraint.mutable=false`). This prevents the attribute from being set through a mutator operation.

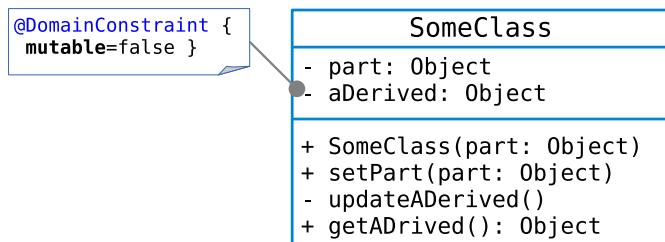


Figure 9.5: Design diagram of derived attribute pattern.

The answer to design question 2 consists of two parts. First, because the update would be invoked multiple times, it should be defined in a helper operation. Second, this helper operation needs to be invoked at the points where the observed parts are changed. These points include the constructor and mutator operations that update the observed parts.

The solution is made clear in design diagram of Figure 9.5. Class `SomeClass` has an attribute named `part`, which represents the observed part, and a derived attribute named `aDerived`. This attribute is derived from `part`. Operationwise, `SomeClass` consists of a constructor and a setter operation, both of which have a parameter named `part`. These are the update points for the attribute `part`. The observer operation `getADerived` is the getter for `aDerived`. It simply returns the value of this attribute. The computation is performed by a separate helper operation named `updateADerived`. This operation is invoked inside the constructor and the setter operations. Note that operation `updateADerived` does not take any parameters, because all the input that it needs are provided by the attributes of the observed parts.

9.6.3 Example

Figure 9.6 shows the UML design of the class `Customer` that has been updated with the pattern solution for the derived attribute `age`. The solution for derived attribute `surname` is defined in a similar manner. Listing 9.3 is a partial code of the class `Customer`, which realises the design in Figure 9.6. The listing includes the declaration of the attribute `age` and how it is computed by the helper operation `updateAge`. This operation is invoked by the constructor and `setDob`, after the value of attribute `dob` has been set. Note, in particular, that attribute `age` is defined with property `DomainConstraint.mutable=false`.

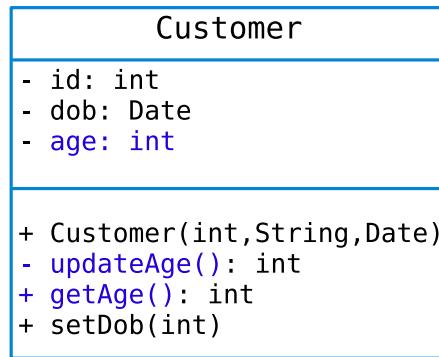


Figure 9.6: Class Customer with derived attribute age.

Listing 9.3: Derived attribute Customer.age

```

1  public class Customer {
2      // code omitted
3      /** a derived attribute */
4      @DomainConstraint(mutable=false)
5      private int age;
6      /**
7       * @effects
8       *   initialise this as Customer:<id,name,dob>
9       *   update age
10      */
11     public Customer(int id, String name, Date dob) {
12         // code omitted
13         this.dob=dob;
14         updateAge();
15     }
16
17     /**
18      * @effects
19      *   (omitted)
20      *   update age
21      */
22     public boolean setDob(Date dob) {
23         // code omitted
24         this.dob = dob;
25         updateAge();
26     }
27
28     /**
29      * @requires dob != null
30      * @effects
31      *   set age = currentYear - year(dob)
  
```

```

32  /*
33  private void updateAge() {
34      Calendar cal = Calendar.getInstance();
35      int currYear = cal.get(Calendar.YEAR);
36      cal.setTime(dob);
37      int yob = cal.get(Calendar.YEAR);
38      age = currYear - yob;
39  }
40 }
```

 **Question** Apply the derived attribute pattern to class *IntSet*.

The following code demonstrates a simple client program that uses *Customer* and its derived attribute *age*:

```

Calendar cal = Calendar.getInstance();
cal.set(1970,0,2);
Date date = cal.getTime();
Customer cust = new Customer(1,"Duc Le", date);
System.out.println(cust);
System.out.println(cust.toString(false));
System.out.println("age: " + cust.getAge());
```

9.6.4 Consequences

The derived attribute pattern has the following benefits:

- **enhanced encapsulation** through making the derived attribute a part of the object state.
- **increased efficiency** through minimising the computation time of the derived attribute value.

However, the pattern has the following disadvantage(s):

- **increased object state complexity** by the introduction of the derived attributes. If not managed well, these attributes would overwhelm the observed parts.

9.7 Recursive Class

The term ‘recursive’ does not actually refer to class but to its objects. A *recursive class* is a class whose objects are defined by a recursive definition. The objects of a recursive class represent the data objects that are captured in the definition. Before

presenting the design pattern for recursive class, let us first briefly review recursive definition.

9.7.1 Background: Recursive Definition

A recursive definition [1] inductively defines class(es) of some objects in terms of the objects themselves. It consists of two types of clauses: basis and induction. The **basis clause** defines the smallest objects. One or more of the **induction clauses** define larger objects in terms of the smaller ones. For example, the following sequences contain objects that are defined by some recursive definitions:

1. 0, 2, 4, 6, ...
2. 0, 1, 3, 6, 10, 15, 21, 28, ...
3. 0, 1, 2, 6, 24, 120, 720, ...



Question What are the underlying recursive definitions of the above sequences?

Example 9.1 Factorial numbers

Listing 9.4 shows the recursive definition of the well-known factorial numbers (sequence 3 in the example above). There is a clear relationship between the successive numbers in this sequence: starting from the base case of number 1, the next factorial equals its index times the preceding factorial. This example will be used in this section to demonstrate the design pattern.

Listing 9.4: Factorial numbers

BASIS.

$$1! = 1.$$

INDUCTION.

$$n! = n \times (n - 1)!.$$

□

9.7.2 Problem

Let's discuss the intent and motivation for recursive class pattern.

Intent

To design a class whose objects are constructed through a recursive definition. This class, called **recursive class**, realises the recursive definition such that it becomes the template for creating the objects.

Motivation

Recursive definition is one of the key constructs used in computer science and software engineering. It is used to define sets, whose elements have some structural relationship. Well-known examples of these sets include data types, strings and grammar rules. For instance, all the strings that are defined based on a given alphabet can recursively be constructed. Many commonly-used natural number subsets are recursive; e.g. even numbers, factorial numbers, fibonacci numbers, prime numbers, and so on. Some important data types that are used in most software, such as list and tree, are recursive in nature. The context-free grammar [28] that is commonly used to define programming language syntax consists of a set of recursively-defined rules.

There are different techniques used to design a recursive definition in a program. A common technique is to use recursive function, which is a function that invokes itself on some pre-defined parameter values. While this design is generally efficient, it arguably does not fully capture the nature of recursive definition. Recursive function merely focuses on computing the final result; it forgets the **state structure** consisting of all the smaller (intermediate) elements that take part in that computation. This state structure is inherent in the recursive definition, due to the repeated application of the inductive clauses. For example, to compute the factorial number $5!$, a recursive function would return the computed result of 120, without recording all the smaller factorials ($1!$, $2!$, $3!$ and $4!$) that take part in the computation.

To fully capture the recursive definition requires a design that takes into account not only the objects but the state structure that defines them.

9.7.3 Solution

Three features of recursive definition that make it suitable for class design. First, the definition concerns a set of objects of a given type. This description fits that of a class. Second, the state structure that needs to be captured is a natural fit for object state. Third, the definition involves constructing objects according to some pre-defined rules. This fits the behaviour of the constructor operation.

The class that realises a recursive definition is called **recursive class**. This class implements the aforementioned features as follows:

- **@object** section of header specification: presents the recursive definition. This definition is what defines the objects.
- **object representation**: includes attributes that capture not only the current object value but also the state structure.

- **private constructor operation(s)** that implement the basis and induction clauses of the recursive definition.
- **static getter** operation that computes and returns an object. This operation looks up in state structure for pre-computed objects and only invokes the constructor if the object is not there.
- **private look-up** operation, which is a helper operation that looks up in the state structure for an object given its index.

An important observation is that since the state structure records all the intermediate objects, it can be used to provide the objects that the client program needs, without creating new ones. This observation is implemented in the design by hiding the constructor operation (making it private) and introducing a **static getter** operation. This operation invokes the helper look-up operation to determine whether or not the desired object has already been created. If so, it immediately returns the object reference; otherwise, it invokes the constructor to create the object before returning it. The constructor realises the recursive definition by creating the current object and recording it into the state structure. It uses the helper look-up operation to determine if smaller objects have been created. If not, it recursively invokes the constructor(s) to create those objects. These in turn place these smaller objects into the state structure.

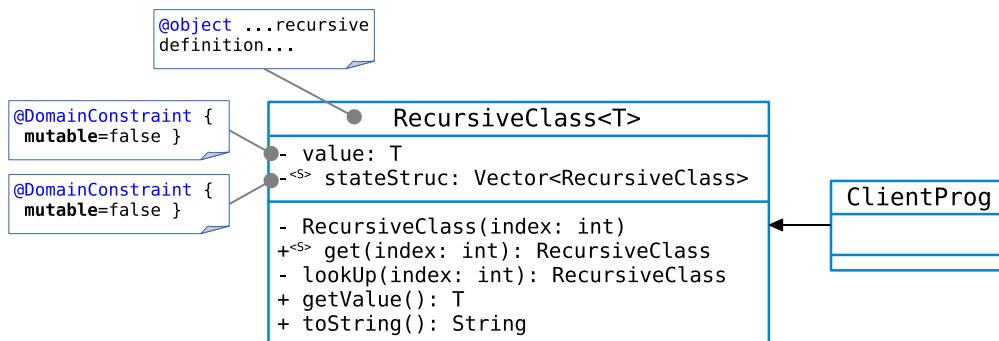


Figure 9.7: Design diagram of recursive class.

Figure 9.7 shows the design diagram of recursive class. To enhance reuse, the class is designed as a generic class, whose type parameter is the data type of object value. The object representation consists of the attribute `value` which records value of the current object. The state structure is represented by a static variable `stateStruc` and is used as a shared buffer for all objects. The data type of this variable is the parameterised `Vector<RecursiveClass>`. Note that both attributes are immutable (i.e. has property `DomainConstraint.mutable = false`), because they cannot be changed directly by the client program. The private constructor operation `RecursiveClass` takes a parameter named `index`. This parameter specifies the index of the object to be constructed.

In practice, other constructors may be added to suit the recursive definition, if needed. Last but not least, the observer operation `getValue` is a minimum that this class needs which provides the object value.

Design alternatives

The inclusion of the state structure in the recursive class is based on an assumption that a single instance of the basis and inductive clauses is shared among all the objects. In practice, however, multiple instances of a clause may exist, which lead to different alternative objects for the same application of this clause. For example, the inductive clause of the recursive definition of string may be applied to any character of the base alphabet and to any of the existing intermediate objects that have been created. Each application thus requires an input character and an intermediate object. This is different from the behaviour of the solution above, where the intermediate objects are created automatically from the same application of the inductive clause and are thus not required as input.

In these cases, the solution should be changed as follows:

- update the constructor operation to take the smaller objects as input
- update the static `getter` operation to match the constructor parameters

9.7.4 Example

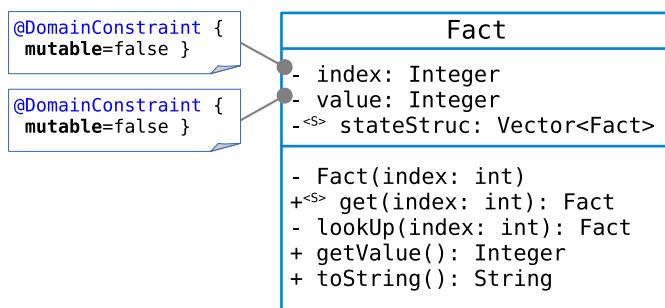


Figure 9.8: The design of recursive class Fact.

Figure 9.8 shows the design of the recursive class named `Fact`. This class realises the recursive definition of factorial numbers presented in Example 9.1. The `Fact` object value is captured by the attribute `value: Integer`. The state structure is captured by the static attribute `stateStruc: Vector<Fact>`. To increase utility, class `Fact` also records the factorial number `index` in the attribute `index`. This is useful if the client program needs to ascertain the position of a `Fact` object in the sequence. The constructor

operation realises both clauses of the recursive definition. It is a recursive function that computes the factorial number specified by `index`.

Listing 9.5 shows the Java code of the class `Fact`. Note how the constructor records all the smaller `Facts` in `stateStruc`. For testing purposes, the look-up operation includes an if statement that prints to the standard output the `Fact` object if it is found in `stateStruc`. This is used to test that `stateStruc` actually works by returning a precomputed object. In production, this debug statement should be commented out.

Listing 9.5: Recursive class `Fact`

```

1 import java.util.Vector;
2 import utils.DomainConstraint;
3 /**
4 * @overview
5 * Represents a factorial number.
6 * @attributes
7 * val Integer
8 * stateStruc Vector<Fact>
9 * @object
10 * A typical factorial is n!, which is defined as follows:
11 * BASIS.
12 * 1! = 1.
13 * INDUCTION.
14 * n! = n x (n - 1)!
15 * @abstract_properties
16 * mutable(val)=false /\ optional(val)=false /\ min(val) = 1 /\ 
17 * mutable(stateStruc)=false /\ 
18 * stateStruc.size() > 0
19 * -> for all e1, e2 in stateStruc. e1 neq e2
20 * @author dmle
21 */
22 public class Fact {
23     private static final int MIN_INDEX = 1;
24     @DomainConstraint(mutable=false, optional=false, min=MIN_INDEX)
25     private Integer index;
26     @DomainConstraint(mutable=false, optional=false, min=1)
27     private Integer val;
28     @DomainConstraint(mutable=false, min=1)
29     private static Vector<Fact> stateStruc;
30
31 /**
32 * @requires n >= 1 /\ stateStruc != null
33 * @effects constructs this to be the nth factorial number
34 */

```

```

35  private Fact(int n) {
36      this.index = n;
37      if (n == 1) {
38          val = 1;
39      } else {
40          // get/compute previous objects
41          Fact prev = lookUp(n-1);
42          if (prev == null) prev = new Fact(n-1);
43          // compute this object
44          val = n * prev.val;
45      }
46      // puts this object into stateStruc
47      stateStruc.add(this);
48  }
49
50 /**
51 * @effects
52 * if index is valid
53 *   create (if not already) and return the factorial number at that index
54 * else
55 *   return null
56 */
57 public static Fact get(int index) {
58     if (index < MIN_INDEX) return null;
59     // get/compute the nth object
60     if (stateStruc == null) stateStruc = new Vector<>();
61     // determine if this number been computed
62     Fact me = lookUp(index);
63     if (me == null) // not yet computed...
64         me = new Fact(index);
65     return me;
66 }
67
68 /**
69 * @effects
70 * if the nth Fact is found in stateStruc
71 *   return it
72 * else
73 *   return null
74 */
75 private static Fact lookUp(int n) {
76     Fact f = null;
77     if (n <= stateStruc.size()) {

```

```

78     f = stateStruc.get(n-1);
79 }
80 // debug
81 if (f != null) System.out.printf("--> Found: %s%n", f);
82 return f;
83 }
84
85 /**
86 * @effects
87 *   return val
88 */
89 public int getValue() {
90     return val;
91 }
92
93 @Override
94 public String toString() {
95     return String.format("Fact(%d!,%d)", index, val);
96 }
97 }
```



Question Design and code recursive classes for the recursive definition examples discussed at the beginning of Section 9.7.1.

The following code demonstrates a simple client program that uses the Fact class to compute some factorial numbers. To test the use of state structure, the numbers are computed in reverse order. The smaller numbers should have already been computed and provided immediately from the state structure.

```

int[] nums = {7, 6, 5, 4, 3, 2, 1};
for (int n : nums) {
    System.out.printf("Computing %d!%n", n);
    Fact f = Fact.get(n);
    System.out.printf(" %s%n", f);
}
```

9.7.5 Consequences

Recursive class pattern has the following benefits:

- ***captures the state structure*** of recursive definition. This state structure holds the components that make up each object state and thus needs to be made explicit in

the design.

- ***improved performance*** in computing the objects, thanks to the shared state structure. Every object is computed only once, either when it is first created or when it is computed as part of a larger object.

However, the pattern has the following disadvantage(s):

- ***requires more memory space*** to hold the state structure. The memory usage is also higher than that of recursive function, because every data element is represented as object.

Chapter 9 Exercise

The exercises in this chapter help practise applying the solutions of the class design patterns to improve a class's design.

1. Change the design and code of class `MobilePhone` of the `GreetingConversation` program of [Exercise 6.1](#) to include an attribute named `colorCode`, which is derived from the attribute `color`. The value of `colorCode` is the standard RGB web color code value, whose color name is the value of `color`. The common RGB web color tables are available at http://www.rapidtables.com/web/color/RGB_Color.htm. For example, if a `MobilePhone` object has `color = 'B'` then its `colorCode = "#0000FF"`.
2. Change the design and code of the class `GreetingConversation` program of [Exercise 6.1](#) so that it becomes a singleton class.
3. Design and implement a recursive class of the numbers in the sequence **1** in the example of Section [9.7.1](#).
4. Design and implement a recursive class of the numbers in the sequence **2** in the example of Section [9.7.1](#).
5. Design and implement a recursive class of the Fibonacci numbers.

Chapter 10

Abstract Data Type:

Object Oriented Design and Implementation

Objectives

- ✓ Understand abstract data type (ADT) and two common ADTs (list and tree).
- ✓ Explain the recursive nature of list and tree and how this helps inform their design.
- ✓ Apply the recursive class design technique to design tree and list.
- ✓ Implement the tree and list designs in Java.

In this chapter, we will discuss the design, implementation of two fundamental data types: **tree** and **list**. These are important data abstractions that are frequently used in designing complex software functions. Figure 10.1 shows the structure of a typical

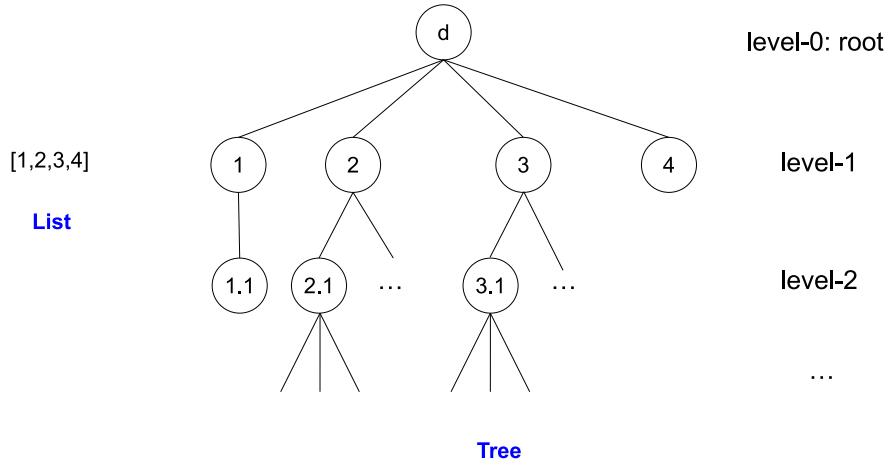


Figure 10.1: An illustration of the list and tree abstractions of a document.

document, which helps illustrate list, tree and the relationship between them. One or the other abstraction applies depending on how one views a document. Skimming through the document, it appears as a list of section headers, one after another in sequence. However, when looking at the document more thoroughly, it appears as a hierarchical structure of headers: the level-1 section headers, the level-2 sub-section headers appearing underneath each level-1 header, the level-3 sub-sub-section headers appearing underneath each level-2 header, and so on. This hierarchical structure is called a tree. At the top level (called level 0) of the document tree in the figure is a node which

represents the document. This is the root node of the tree and is also the only node at this level. This node is linked to four level-1 nodes that are labelled 1, 2, 3, and 4. These represent the four level-1 section headers of the document. Each level-1 node is in turn linked to one or more level-2 nodes and so on.

This chapter first gives an overview of list and tree, emphasising the recursive nature of both abstractions. After that, it applies the recursive class design pattern to define the design of each data type.

10.1 Overview

This section gives an overview of abstract data type and discusses list and tree from the perspective of recursion. It also briefly discusses Java's support for list and tree and the design approach of this book.

10.1.1 Abstract Data Type

Abstract data type (ADT) is a data type that models an abstract concept in a problem domain. There are frequently-used ADTs that are fundamental to many domains. These ADTs solve frequently-occurring problems which are similar in character to those tackled by design patterns. The difference between ADT and design pattern lies only in the problem scope. ADTs are used directly as data types in a software, while design patterns involves combining several data types. ADTs target relatively smaller design problems (e.g. a sequence of items or a hierarchical structure). The problems that design patterns address are often broader, the solutions of which may involve ADTs. For example, the recursive class design pattern (discussed in the previous chapter) uses a collection data type, named `Vector`, as part of its solution. In Java, `Vector` is a type of list.

Thus, ADTs help improve the development productivity of software. This book focuses on two important ADTs: list and tree.

10.1.2 List and Tree

It is worth highlighting the following properties of list and tree, which help reveal the nature of and the relationship between these two data types:

- list and tree are recursive data structures
- tree is a type of recursive list
- list is a type of binary tree

As shown earlier in the document structure example of Figure 10.1, a list represent a sequence of items, while a tree represents a hierarchical structure of items. Further, there is a relationship between list and tree. Each level of the tree consists of lists of nodes. Conceptually, therefore, a tree can be viewed as a ‘recursive’ list, elements of which are themselves lists. A list can be represented as a binary tree as shown in Figure 10.2. The

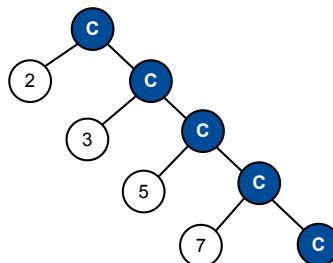


Figure 10.2: A binary-tree representation of the list (2,3,5,7).

figure demonstrates with a sample list (2,3,5,7). The filled circles labelled ‘C’ represent *constructor* nodes, which are artificial nodes added to construct the overall tree structure. The effect of this structure is that the list elements are arranged in the same sequential order of the list. Each constructor nodes has two branches: the left branch contains just an element, the right branch is another constructor node. The last constructor node terminates the tree with no further elements to have.

10.1.3 Java Support

List is implemented as part of the Java’s collection API (package: `java.util`¹). However, Java does not provide an implementation for tree. This means that developer needs to either use a third-party library or provide their own implementation of this data type. In this chapter, we will study the design and implementation of both list and tree. Our approach differs from Java in that it is based on recursion.

Listing 10.1 shows a simple Java code that demonstrates how to use two concrete Java implementations of list: `ArrayList` and `LinkedList`. Since the code to use these two classes differ only in the object creation step, the listing creates a general `List` object (variable `list`), which can be assigned to an object of `LinkedList` or an object of `ArrayList`.

Listing 10.1: Java’s List classes

```

1 import java.util.ArrayList;
2 import java.util.LinkedList;
3 import java.util.List;
  
```

¹<https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html>

```

4 public class JavaArrayListApp {
5     public static void main(String[] args) {
6         // create a List object
7         List<Integer> list = new LinkedList<>();
8         // alternative: list = new ArrayList<>();
9
10        // add/remove some elements to the List
11        int[] elements = {2, 0, 2, 0, 1, 9, 9, 9};
12        for (int x : elements) {
13            list.add(x);
14        }
15        // process the elements in some way...
16        System.out.println(list.getClass().getSimpleName() + ": " + list);
17    }
18 }
```

10.2 Overall Design Approach

The underlying design principle for both Tree and List is **recursion**. For both ADTs, we apply the development approach discussed so far in this book as follows:

1. Define the abstract concept, which includes abstract properties and operations
2. Construct the class design based on an adaptation of the recursive class pattern
3. Implement the design in Java

The recursive design of both ADTs is an adaptation of the recursive class pattern (discussed in the previous chapter). Although the smaller objects still form the state structure of a larger one, they are not reused to create different objects. For example, although the two lists (1,2,3,4) and (1,2,3,5) contain the sequence of 3 elements 1, 2, 3, two separate instances of this sequence are used to construct the two lists.

An important implication of this is that there is no need to use the state structure to record the intermediate objects. The design can thus also be simplified to exclude the static getter and the look-up operations. The private constructor(s) are now made public so that they can be invoked directly by the client program.

10.3 Tree

We focus on a common type of tree called **rooted tree** [1].

10.3.1 Abstract Concept

A rooted tree basically consists of a set of nodes and edges, exactly one of the nodes is the root.

Definition 10.1. Rooted tree

A (rooted) tree consists of a set of nodes and a set of edges, such that:

- *a node, called **root**, is distinguished from other nodes*
- *every edge connects two distinct nodes (binary edge)*
- *every node c other than root is connected by an edge to one other node p:*
 - *c is a child of p, p is the parent of c*
- *tree is **connected**, i.e. root is reachable from any other node by a sequence of edges*



The root of the rooted tree is usually drawn at the top.

Example 10.1 Tree

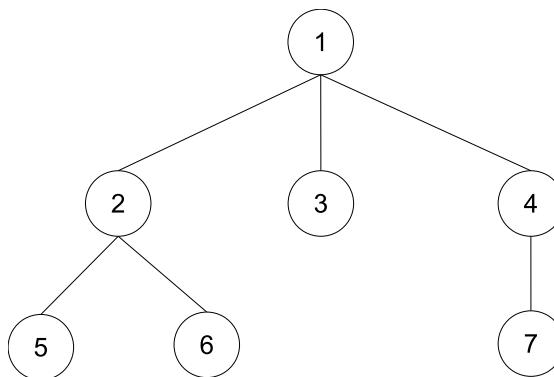


Figure 10.3: Tree example.

Figure 10.3 shows an example of a rooted tree with seven nodes. For the sake of the example, each node is labelled with a node id. In practice, each node can represent an object of any type; the node id becomes the object id. As shown in the figure, node 1 is the root node. The tree has 6 edges. For instance, (2,5) is a binary edge that connects node 2 to node 5. Node 2 is the parent node and node 5 is its child. Besides node 5, node 2 also has node 6 as its child. □

Tree-Related Terms

Among the common tree-related terms [1] are sibling, path, ancestor, descendant, subtree, leaf and interior nodes, height and depth.

Definition 10.2. Sibling

*Nodes that have the same parent are called **siblings**.*



For example, in Figure 10.3, nodes 1, 3, and 4 are siblings; so are nodes 5 and 6.

Definition 10.3. Path

*A sequence of nodes of a tree is a **path** if every node in the sequence, except the last node, is the parent of the subsequent node.*

A path's length is the number of edges in the path.



For example, In Figure 10.3, node sequence (1, 2, 6) form a path of length 2 from the root 1 to node 6. Node sequence (1) is a path of length 0 from 1 to itself.

The parent-child relationship can be extended naturally to ancestors and descendants.

Definition 10.4. Ancestor and descendant

- The **ancestors** of a node are found by following the unique path from the node to its parent, to its parent's parent, and so on.
- The **descendant relationship** is the inverse of the ancestor relationship. That is, a node is a **descendant** of another node if and only if this node is an ancestor the node.



It easily follows from Definition 10.4 that a node is also its own ancestor. Further, the root is an ancestor of every node in a tree and every node is a descendant of the root.

Definition 10.5. Subtree

*In a tree T , a node n , together with all of its proper descendants (if any) is called a **subtree** of T . Node n is the root of this subtree.*



For example, in Figure 10.3, node 3 is a single-node subtree, because this node has itself as a descendant. Nodes 2, 5, and 6 together with the edges connecting them form a subtree. Node 2 is the root of this subtree; nodes 5, 6 are all the descendants of 2. The entire tree of Figure 10.3 is a subtree of itself. Note, however, that nodes 2 and 6 and the binary edge (2,6) cannot form a subtree, because this does not include another descendant of node 2 (node 5).

Definition 10.6. Leaf and interior nodes

- A **leaf** is a node of a tree that has no children.
- An **interior node** is a node that has one or more children.



Thus, the root of a tree that has more than two nodes is an interior node. The root of a single-node tree is both the root and a leaf, but not an interior node. Generally, every

node of a tree is either a leaf or an interior node, but not both.

For example, in Figure 10.3, leave nodes are 5, 6, 3, and 7. Interior nodes are 1, 2, and 4.

Definition 10.7. Height and Depth

- The **height** of a node is the length of a longest path from it to a leaf. The height of the tree is the height of the root.
- The **depth**, or **level**, of a node is the length of the path from the root to it.



For example, in Figure 10.3, node 1 has height 2, node 2 has height 1, and the leaf node 3 has height 0. The tree's height is 2. The depth of node 1 is 0, the depth of node 2 is 1, and the depth of node 5 is 2.

Recursive Definition

There are two recursive definitions for rooted tree. Each definition gives rise to a different tree design:

- **bottom-up**: tree is formed from the sub-trees
- **top-down**: tree is formed by extending out from the root

In the bottom-up view, a new (larger) tree is constructed from two more smaller ones by adding suitable edges that connect them. In the top-down view, a new tree is constructed from an existing tree by adding new edges that connect a node of this tree to some other new nodes. Let us attempt to formalise the two definitions. The definitions use the tree notation introduced in Definition 10.8.

Definition 10.8. Tree

A tree, denoted by T , is a triple $\langle r, \text{nodes}, \text{edges} \rangle$, where:

- nodes is the set of nodes
- edges is the set of edges
- r is the root ($r \in \text{nodes}$)

To ease notation, we denote:

- $T.\text{root} = r$, $T.\text{nodes} = \text{nodes}$, $T.\text{edges} = \text{edges}$.
- $\text{edge}(p, n)$ is an edge that connects the parent node p to the child node n .



Bottom-Up Definition

Definition 10.9. Bottom-up Tree

Basis.

$\forall \text{node } r, T = \langle r, \{r\}, \emptyset \rangle \text{ is a tree.}$

Induction.

$\forall 1 \leq k \in \mathbb{N}, \forall T_1, \dots, T_k \in \text{Tree} \text{ and } \forall \text{node } r \notin T_1, \dots, T_k:$

$$T = \langle r, T_1.\text{nodes} \cup \dots \cup T_k.\text{nodes} \cup \{r\},$$

$T_1.\text{edges} \cup \dots \cup T_k.\text{edges} \cup \{\text{edge}(r, T_1.\text{root}), \dots, \text{edge}(r, T_k.\text{root})\} \rangle$
is a tree.



Example 10.2 Bottom-up tree

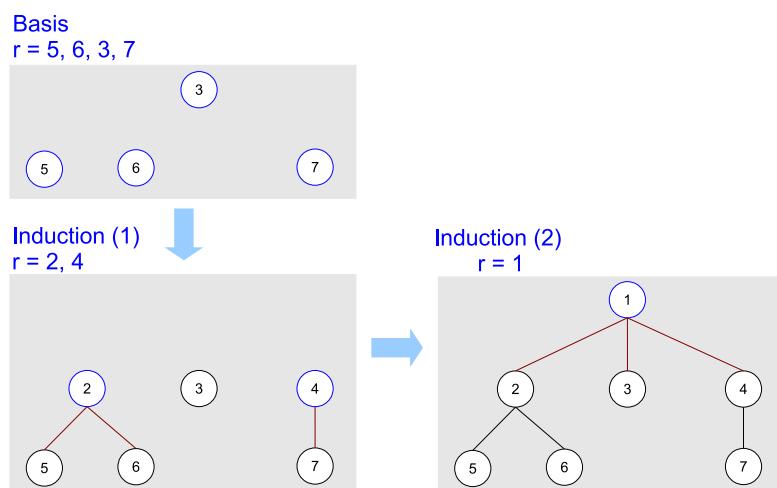


Figure 10.4: Constructing the tree in Figure 10.3 from the bottom up.

Figure 10.4 demonstrates how to construct the tree in Figure 10.3 by applying the bottom-up recursive definition. First the basis clause is applied to create the four single-node trees $T_3 = \langle 3, \{3\}, \emptyset \rangle$, $T_5 = \langle 5, \{5\}, \emptyset \rangle$, $T_6 = \langle 6, \{6\}, \emptyset \rangle$ and $T_7 = \langle 7, \{7\}, \emptyset \rangle$. Then the induction clause is applied three times to result in the final tree. First, it is applied to node 2 and trees T_5, T_6 to create a new tree T_2 . Second, it is applied to node 4 and tree T_7 to create another new tree T_4 . Third, it is applied to node 1 and the three trees T_2, T_3 and T_4 to produce the target tree. \square

Top-Down Definition

Definition 10.10. Top-down Tree

Basis.

$\forall \text{ node } r, T = \langle r, \{r\}, \emptyset \rangle \text{ is a tree.}$

Induction.

$\forall \text{ node } n, \forall T' \in \text{Tree} \text{ that } n \notin T' \text{ and for some node } p \in T':$

$T = \langle T'.\text{root}, T'.\text{nodes} \cup \{n\}, T'.\text{edges} \cup \{\text{edge}(p, n)\} \rangle$

is a tree.



Example 10.3 Top-down tree

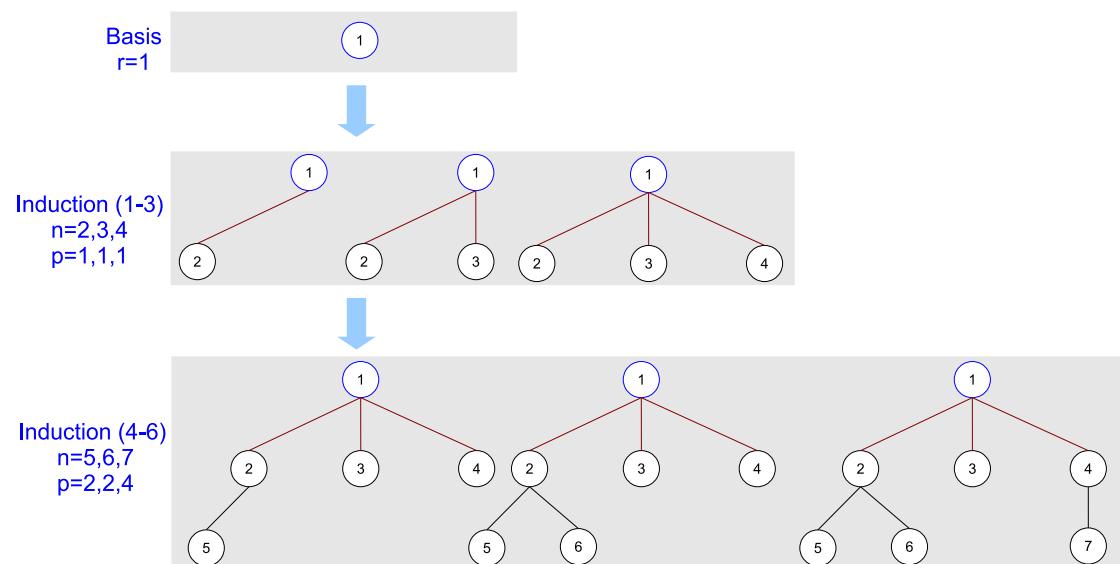


Figure 10.5: Constructing the tree in Figure 10.3 from the top down.

Figure 10.5 demonstrates how to construct the tree in Figure 10.3 by applying the top-down recursive definition. The application consists in following sequence of steps:

1. basis clause on node 1 creates tree $T_1 = \langle 1, \{1\}, \emptyset \rangle$
2. induction clause on $T' = T_1$, $n = 2$, $p = 1$
3. induction clause on T' , $n = 3$, $p = 1$
4. induction clause on T' , $n = 4$, $p = 1$
5. induction clause on T' , $n = 5$, $p = 2$
6. induction clause on T' , $n = 6$, $p = 2$
7. induction clause on T' , $n = 7$, $p = 4$

□

Abstract Properties

First, a rooted tree must have exactly one node that serves as the root. Secondly, a rooted tree shares a general property of a tree, which is an acyclic, connected graph. This means that there exists exactly one path that connects any two nodes of the tree. Putting these together give us three rooted tree properties [1]. These properties form the abstract properties of the tree design.

Definition 10.11. Tree properties

Rooted tree has the following properties:

1. *root node must be specified.*
2. *(connectedness) there exists a path from root to every non-root node.*
3. *(acyclicity) every non-root node has exactly one parent.*



10.3.2 Operations

Since tree is recursive, the operations performed on tree are inherently recursive. In general, the behaviour of an operation consists of two parts. The first part is a recursion and the second part is the actual computation that is performed against the recursion.

The recursion part helps traverses the tree in a top-down manner, one node at a time, until it reaches the leafs. The basis case is applied at the leaf nodes while the induction is applied at the non-leaf ones.

The computation part defines what to do at each node. The logic the this computation depends on the application. Below is a list of the computations, adapted from [1], that are believed to be common to all trees:

- **height:** returns the height of the tree or a subtree
- **countif:** counts the number of nodes of the tree or a subtree that satisfy a condition
- **count:** (a special form of countif) counts the number of nodes of the tree or a subtree
- **eval:** evaluates the expression that is represented by the tree
- **toString:** turns a tree or a subtree into a structured string

This section will focus on two operations `count` and `toString`. Details of the other operations are given in [1].

Operation count

Definition 10.12. Operation count

Operation count, invoked on some node n that has k children c₁, … c_k (k ≥ 0), is defined as follows:

$$\text{count}(n) = \begin{cases} 1 & \text{if } n \text{ is leaf} \\ 1 + \sum_{c_i} \text{count}(c_i) & \text{if } n \text{ is not leaf} \end{cases}$$



Example 10.4

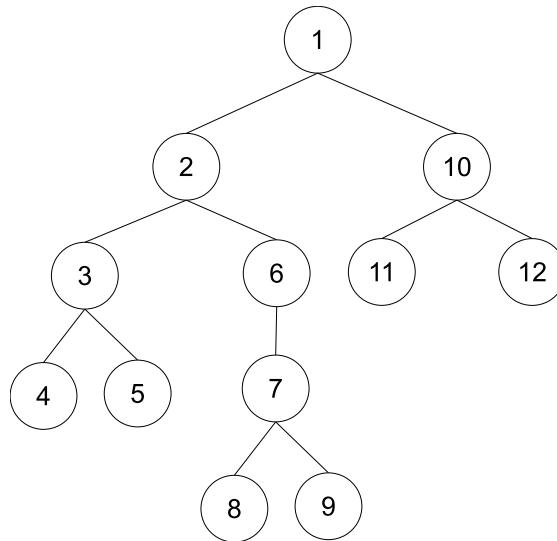


Figure 10.6: A rooted tree with 12 nodes.

Let us consider a bigger tree in Figure 10.6 which has 12 nodes and 11 edges. In this tree, it is clear that $\text{count}(7) = 3$, which covers node 7 itself and its two children. As another example, $\text{count}(2) = 8$, which covers all the nodes of the subtree rooted at node 2. \square

Operation `toString`

A structured string representation of a tree involves using a number of separator symbols and some utility operations on these symbols. These are described as follows:

- NIL is the empty string ("")
- NL is the next line character
- VER is the vertical bar character ("|")
- operation spaceMarkers(l) = $\begin{cases} \text{NIL} & \text{if } l = 0 \\ "\dots" & \text{if } l > 0 \end{cases}$

- operation $\text{bol}(l) = \begin{cases} \text{NIL} & \text{if } l = 0 \\ \text{NL} + \text{VER} & \text{if } l > 0 \end{cases}$
- operation $\text{nodeLine}(n) = \text{bol}(n.\text{level}) + \text{spaceMarkers}(n.\text{level}) + n.\text{toString}$

Definition 10.13. Operation `toString`

Operation `toString`, invoked on some node n at some level l ($0 \leq l \in \mathbb{N}$) that has k children c_1, \dots, c_k ($k \geq 0$), is defined as follows:

$$\text{toString}(n) = \begin{cases} \text{nodeLine}(n) & \text{if } n \text{ is leaf} \\ \text{nodeLine}(n) + \\ \quad \text{toString}(c_1) + \dots + \text{toString}(c_k) & \text{if } n \text{ is not leaf} \end{cases}$$



Example 10.5

```

1
|-2
|--3
|---4
|---5
|---6
|---7
|---8
|---9
|-10
|--11
|--12

```

Listing 10.5 shows the result of performing `toString(1)` on the tree in Figure 10.6. It displays the structured string representation of the entire tree. \square

10.3.3 Design Approach

There are different ways to design tree. For instance, Aho [1] discusses using an array or list-based data structure to represent the tree structure. In this section, we will discuss an alternative design using recursive class (discussed in the previous chapter). This design, though not necessarily always the most efficient, is arguably more natural as it appeals to the tree's recursive nature. The following paragraphs describe the key design questions concerning the state and behaviour spaces.

Verifying the abstract properties

The abstract properties stated in Definition 10.11 can easily be verified programmatically. Property 1 is a simple null check on the root node. Property 2 can be verified using the nodes and edges sets. Property 3 can be verified using the operation count. If the result of this count is smaller than the cardinality of the nodes set then we know that the tree is not connected.

How to represent a node?

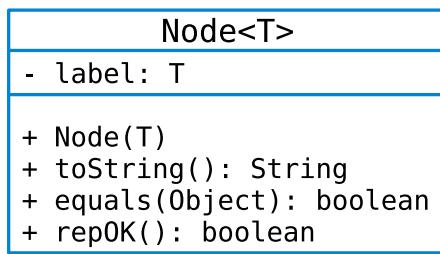


Figure 10.7: Class Node.

Nodes are objects of a generic class named `Node<T>`. The type parameter `T` represents the data type of the node label. Figure 10.7 shows the design of this class. It is the most basic form, which contains one attribute called `label`. Other attributes can easily be added to capture more complex node data if needed. Appendix 10.A presents the design specification and Java code of class `Node`.

How to represent an edge?

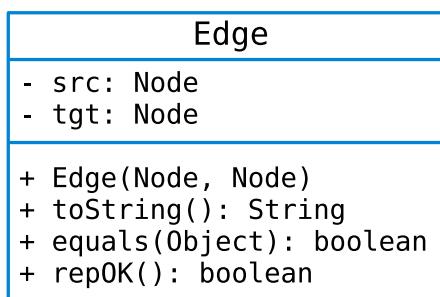


Figure 10.8: Class Edge.

Edges are objects of a general class named `Edge`. Figure 10.8 shows the design of this class. It is in its most basic form, which has two attributes: `src` and `tgt`. The former more represents the source node (which is a parent) and the latter represents the target node (which is a child). Similar to `Node`, class `Edge` can easily be extended with other

attributes to capture more complex edge data (e.g. label, weight, *etc.*). Appendix 10.B presents the design specification and Java code of class Node.

How to represent sets of nodes and edges?

Similar to the class IntSet (discussed in an earlier chapter), nodes and edges are represented by the generic class Vector<T>. More specifically, nodes set is represented by Vector<Node> and edges set is by Vector<Edge>. This is more productive and efficient than array since it allows us to dynamically update the tree as nodes and edges are added to (and also potentially removed from) it.

10.3.4 Bottom-Up Design

Figure 10.9 shows the recursive, bottom-up design of the class Tree.

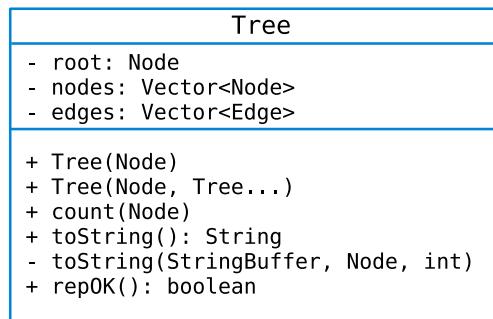


Figure 10.9: The recursive, bottom-up design of class Tree.

Constructors

Class Tree in Figure 10.9 has two constructors which realise the two clauses of the recursive definition (Definition 10.9). The one-argument constructor realises the basis clause, while the other constructor realises the inductive one.

Operation count

This operation realises the recursive Definition 10.12 by traversing the subtree rooted at the input node and counting the number of visited nodes.

Operation toString

This operation realises the recursive Definition 10.13 by recursively traversing the tree starting at the input node and constructs a string consisting of all the nodes that are visited.

Design Specification

Listing 10.2 shows the design specification of the bottom-up class Tree.

Listing 10.2: Design specification of bottom-up Tree

```

1 import java.util.Vector;
2 import chap10.tree.Edge;
3 import chap10.tree.Node;
4 import utils.DomainConstraint;
5
6 /**
7 * @overview A tree is a set of nodes that are connected to each other by
8 * edges such that one node, called the root, is connected to some nodes,
9 * each of these nodes is connected to some other nodes that have not been
10 * connected, and so on.
11 *
12 * <p>The following is a <b>bottom-up</b> recursive tree design that
13 * incrementally builds a new tree by adding roots.
14 *
15 * @attributes
16 *   root   Node
17 *   nodes  Set<Node>           Vector<Node>
18 *   edges  Set<Edge>  Vector<Edge>
19 *
20 * @object A typical tree T is the tuple <r,N,E>, where
21 *   root(r), nodes(N), and edges(E).
22 *
23 * <p>Trees are defined recursively as follows:
24 * Basis
25 *   For any node r, T = <r,{r},{}> is a tree.
26 * Induction
27 *   For all natural number k >= 1 and k trees T1, T2,...,Tk
28 *   and for all node r:
29 *     r is not in T1,...,Tk ->
30 *       T = <r, T1.nodes+...+Tk.nodes+{r},
31 *             {edge(r,T1.root),...,edge(r,Tk.root)} +T1.edges+...+Tk.edges>
32 *   is a tree
33 *
34 * @abstract_properties <pre>
35 *   mutable(root)=false /\ optional(root)=false /\
36 *   mutable(nodes)=true /\ optional(nodes)=false /\
37 *   mutable(edges)=true /\ optional(edges)=true /\
38 *   root in nodes /\
39 *   for all n in nodes.

```

```

40 *      // (i) acyclicity
41 *      (exists p in nodes. parent(n)=p /\ 
42 *          for all p, q in nodes. (parent(n)=p /\ parent(n)=q -> p=q)) /\ 
43 *          // (ii) connectedness
44 *          for all n in nodes.d
45 *              (n != root -> exists a sequence of edges from root to n)
46 * </pre>
47 *
48 * @author dmle
49 */
50 public class Tree {
51     @DomainConstraint(type="Node",mutable=false,optional=false)
52     private Node root;
53     @DomainConstraint(type="Vector",mutable=true,optional=false)
54     private Vector<Node> nodes;
55     @DomainConstraint(type="Vector",mutable=true,optional=true)
56     private Vector<Edge> edges;
57
58     // constructors
59 /**
60     * @requires r != null
61     * @effects initialise this as <r,{r},{}>
62 */
63     public Tree(Node r)
64
65 /**
66     * @requires r != null /\ trees.length >= 1 /\
67     * for all t in trees. r not in t.nodes
68 *
69     * @effects initialise this as a tree T =
70     *      <r,
71     *      T1.nodes+...+Tk.nodes+{r},
72     *      {edge(r,T1.root),...,edge(r,Tk.root)} +T1.edges+...+Tk.edges>,
73     *      where Tis are in <tt>trees</tt>
74 */
75     public Tree(Node r, Tree...trees)
76
77 /**
78     * A recursive procedure to count the number of nodes in a subtree
79     * rooted at n.
80     *
81     * @effects
82     * if n is a leaf

```

```

83     *      return 1
84     * else
85     *      return the number of nodes in the sub-tree rooted at n
86     */
87     public int count(Node n)
88
89 /**
90  * @effects
91  *  return a structured textual representation of this
92  */
93 @Override
94 public String toString()
95
96 /**
97  * @effects
98  * if this satisfies abstract properties
99  * return true
100 * else
101 * return false
102 */
103 public boolean repOK()
104 } // end Tree

```

10.3.5 Top-Down Design

It is easy to see that the top-down tree design is similar to the bottom-up one except for the operation that implements the inductive rule of the recursive definition. This is where the two definitions differ.

Listing 10.3: The constructor of the top-down Tree

```

1 /**
2  * @requires t != null /\ n is in t.nodes /\ trees.length >= 1 /\ 
3  *   for all t' in trees. (t.nodes - t'.nodes = t.nodes)
4  *
5  * @effects initialise this as a tree T =
6  *   <t.root,
7  *   t.nodes+t1.nodes+...+tk.nodes,
8  *   {edge(n,t1.root),...,edge(n,tk.root)}+t1.edges+...+tk.edges>,
9  *   where tis are in trees
10 */
11 public Tree(Tree t, Node n, Tree...trees)

```

As shown in Definition 10.10, the general form of the inductive rule of the top-down definition appends one or more trees to an existing node of an existing tree. This could be translated directly into the design, resulting in a constructor shown in Listing 10.3. This constructor is similar to the second constructor of the bottom-up design shown in Listing 10.2.

However, a draw-back of the above design is that tree objects need to be created before hand and passed in as argument. This is rather cumbersome and not efficient. We thus propose an alternative design that overcomes this draw-back.

A more efficient design of the inductive rule

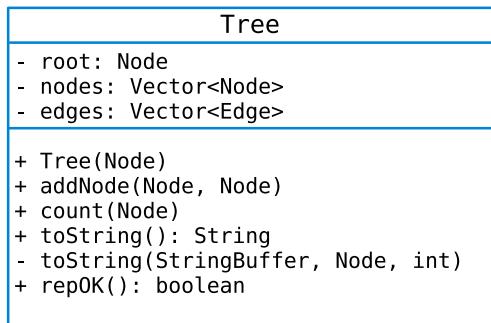


Figure 10.10: The recursive, top-down design of class Tree.

Figure 10.10 shows the recursive, top-down design of class Tree. In this design, the second constructor is replaced by an operation named `addNode`. This operation adds a new node `n` as a child of an existing node `parent` in a tree. Listing 10.4 shows the partial design specification of class Tree that contains the operation `addNode`. Compared to the pure design in Listing 10.3, this design basically treats the current tree object as the first parameter and the parameter `parent` as the second parameter. As for the third parameter, which is a Tree array, it is replaced by a single node, `n`, which serves as the root node of each of the Tree objects in the array. This is because one tree can be added at a time to the existing tree by creating an edge to its root node.

Listing 10.4: Partial specification of the recursive, top-down Tree with operation `addNode`

```

1 import java.util.Vector;
2 import chap10.tree.Edge;
3 import chap10.tree.Node;
4 import utils.DomainConstraint;
5
6 /**
7 * @overview A tree is a set of nodes that are connected to each other by
8 * edges such that one node, called the root, is connected to some nodes,

```

```

9  *      each of these nodes is connected to some other nodes that have not been
10 *      connected, and so on.
11 *
12 *      <p>The following is a <b>top-down</b> recursive design that
13 *      incrementally build a
14 *      tree by adding leaf nodes.
15 *
16 *      @attributes
17 *      root    Node
18 *      nodes   Set<Node>  Vector<Node>
19 *      edges   Set<Edge>  Vector<Edge>
20 *
21 *      @object A typical tree T is the tuple <r,N,E>, where
22 *      root(r), nodes(N), and edges(E).
23 *
24 *      <p>Trees are defined recursively as follows:
25 *      Basis
26 *      For any node r, T = <r,{r},{}> is a tree.
27 *      Induction
28 *      For all node n and tree T' and for some node p in T':
29 *      n is not in T' ->
30 *          T = <T'.root, T'.nodes+{n}, T'.edges+{edge(p,n)}> is a tree
31 *
32 *      @abstract_properties <pre>
33 *          mutable(root)=false /\ optional(root)=false /\
34 *          mutable(nodes)=true /\ optional(nodes)=false /\
35 *          mutable(edges)=true /\ optional(edges)=true /\
36 *          root in nodes /\
37 *          for any n,m in nodes:
38 *              (exists exactly one sequence edge(n,n1),edge(n1,n2),...,edge(nk,m))
39 *      </pre>
40 *
41 *      @author dmle
42 */
43
44 public class Tree {
45     @DomainConstraint(type="Node",mutable=false,optional=false)
46     private Node root;
47     @DomainConstraint(type="Vector",mutable=true,optional=false)
48     private Vector<Node> nodes;
49     @DomainConstraint(type="Vector",mutable=true,optional=true)
50     private Vector<Edge> edges;
51
52     // constructors

```

```

51  /**
52   * @requires r != null
53   * @effects initialise this as <r,{r},{}>
54   */
55  public Tree(Node r)
56
57  /**
58   * @requires <tt>n != null \wedge parent != null</tt>
59   * @effects
60   * if <tt>parent</tt> is in <tt>nodes</tt>
61   * add <tt>n</tt> as a child of <tt>parent</tt>, i.e. <tt>edge(parent,n)</tt>
62   * return true
63   * else
64   * return false
65   */
66  public boolean addNode(Node parent, Node n)
67
68  // other methods (same as the bottom-up Tree class)
69 }

```

10.3.6 Implementation

Bottom-up Design

Listing 10.5 shows the Java implementation of the bottom-up class Tree.

Listing 10.5: Java implementation of bottom-up Tree

```

1 import java.util.Vector;
2 import chap10.tree.Edge;
3 import chap10.tree.Node;
4 import utils.DomainConstraint;
5 /**
6  * @overview ...
7  *
8  * @attributes
9  * root Node
10 * nodes Set<Node> Vector<Node>
11 * edges Set<Edge> Vector<Edge>
12 *
13 * @object ...
14 *
15 * @abstract_properties ...

```

```

16  *
17  * @author dmle
18 */
19 public class Tree {
20     @DomainConstraint(type="Node",mutable=false,optional=false)
21     private Node root;
22     @DomainConstraint(type="Vector",mutable=true,optional=false)
23     private Vector<Node> nodes;
24     @DomainConstraint(type="Vector",mutable=true,optional=true)
25     private Vector<Edge> edges;
26
27     // constructors
28     /**
29      * @requires r != null
30      * @effects initialise this as <r,{r},{}>
31     */
32     public Tree(Node r) {
33         // single-node tree
34         this.root = r;
35         nodes = new Vector<>();
36         nodes.add(r);
37     }
38
39     /**
40      * @requires r != null /\ trees.length >= 1 /\
41      *   for all t in trees. r not in t.nodes
42      *
43      * @effects initialise this as a tree T =
44      *   <r,
45      *   T1.nodes+...+Tk.nodes+{r},
46      *   {edge(r,T1.root),...,edge(r,Tk.root)} +T1.edges+...+Tk.edges>,
47      *   where Tis are in <tt>trees</tt>
48      */
49     public Tree(Node r, Tree...trees) {
50         this(r);
51         edges = new Vector<>();
52         // this.r = r /\ this.nodes = {r} /\ this.edges = {}
53         Edge e;
54         for (Tree t: trees) {
55             nodes.addAll(t.nodes);
56             // this.nodes = this.nodes + t.nodes
57             e = new Edge(r,t.root);
58             edges.add(e);

```

```

59     // this.edges = this.edges + {edge(r,t.root)}
60     if (t.edges != null)
61         edges.addAll(t.edges);
62     }
63 }
64
65 /**
66 * A recursive procedure to count the number of nodes in a subtree
67 * rooted at n.
68 *
69 * @effects
70 *   if n is a leaf
71 *     return 1
72 *   else
73 *     return the number of nodes in the sub-tree rooted at n
74 */
75 public int count(Node n) {
76     int count = 1; // includes n
77     if (edges != null) {
78         for (Edge e : edges) {
79             if (e.hasSrc(n)) { // e[0] is parent
80                 count += count(e.getTgt()); // e[1]: child of n (recursive call)
81             }
82         }
83     }
84     return count;
85 }
86
87 /**
88 * @effects
89 *   return a structured textual representation of this
90 */
91 @Override
92 public String toString() {
93     StringBuffer sb = new StringBuffer();
94     toString(sb, 0, root);
95     return sb.toString();
96 }
97
98 /**
99 * This is a recursive operation that incrementally constructs the string
100 * representation of the
101 * tree/subtree rooted at n.

```

```

101  *
102  * @modifies sb
103  *
104  * @effects
105  * update <tt>sb</tt> with the string representation of the sub-tree
106  * rooted at <tt>n</tt>.
107  */
108 private void toString(StringBuffer sb, int level, Node n) {
109     // append next line
110     if (level > 0) {
111         sb.append("\n");
112         sb.append("|");
113     }
114     for (int i = 0; i < level; i++) {
115         sb.append("-");
116     }
117     sb.append(n);
118     if (edges != null) {
119         int thisLevel = level;
120         for (Edge e : edges) {
121             if (e.hasSrc(n)) { // append this child of n
122                 if (thisLevel == level)
123                     level++;
124                 toString(sb, level, e.getTgt()); // recursive call to next level
125             }
126         }
127     }
128 }
129 /**
130 * @effects
131 * if this satisfies abstract properties
132 * return true
133 * else
134 * return false
135 */
136 public boolean repOK() {
137     // root != null
138     if (root == null) {
139         System.err.println("Tree.repOK: root is null");
140         return false;
141     }
142     // nodes != null

```

```

144     if (nodes == null) {
145         System.err.println("Tree.repOK: nodes is not initialised");
146         return false;
147     }
148     // root in nodes
149     boolean hasRoot = false;
150     for (Node n : nodes) {
151         if (n.equals(root)) {
152             hasRoot=true;
153             break;
154         }
155     }
156     if (!hasRoot) {
157         System.err.println("Tree.repOK: tree does not contain root");
158         return false;
159     }
160     // for all n, m in nodes. n neq m
161     Node n; Edge eobj;
162     for (int i = 0; i < nodes.size(); i++) {
163         n = nodes.get(i);
164         for (int j = i+1; j < nodes.size(); j++) {
165             if (n.equals(nodes.get(j))) {
166                 System.err.println("Tree.repOK: duplicate node: " + n);
167                 return false;
168             }
169         }
170     }
171     // for all e, f in edges. e neq f
172     if (edges != null) {
173         for (int i = 0; i < edges.size(); i++) {
174             eobj = edges.get(i);
175             for (int j = i+1; j < edges.size(); j++) {
176                 if (eobj.equals(edges.get(j))) {
177                     System.err.println("Tree.repOK: duplicate edge: " + eobj);
178                     return false;
179                 }
180             }
181         }
182     }
183
184     // check: every non-root node has exactly one parent
185     Node parent;
186     for (Node o : nodes) {

```

```

187     parent = null;
188     if (o != root) {
189         for (Edge e : edges) {
190             if (e.hasTgt(o)) {
191                 if (parent == null) {
192                     parent = e.getSrc();
193                 } else {
194                     // invalid: two parents
195                     System.err.println("Tree.repOK: node has two parents: " + o +
196                         "-> (" + parent + "," + e.getSrc() + ")");
197                     return false;
198                 }
199             }
200         }
201         if (parent == null) {
202             // invalid: no parents
203             System.err.println("Tree.repOK: node has no parents: " + o);
204             return false;
205         }
206         // o has one parent
207     }
208 }
209
210 // all non-root nodes are reachable from n (i.e. tree is connected)
211 // walk the tree from the root and count the number of nodes
212 // check that this number is the same as cardinality of nodes
213 int count = count(root);
214 if (count != nodes.size()) {
215     System.err.println("Tree.repOK: tree is not connected");
216     return false;
217 }
218 return true;
219 }
220 } // end Tree

```

The program class that uses the bottom-up Tree class is `chap10.test.tree.BottomUpTreeTest` in the attached source code. Listing 10.6 shows the code of this class.

Listing 10.6: An example program class for the bottom-up Tree

```

1 import chap10.tree.Node;
2 import chap10.tree.bottomup.Tree;
3 public class BottomUpTreeTest {

```

```

4  public static void main(String[] args) {
5      Tree t4 = new Tree(new Node(4)), t5 = new Tree(new Node(5));
6      Tree t8 = new Tree(new Node(8)), t9 = new Tree(new Node(9)),
7          t7 = new Tree(new Node(7),t8,t9);
8      Tree t11 = new Tree(new Node(11)),
9          t12 = new Tree(new Node(12)),
10     t10 = new Tree(new Node(10),t11,t12);
11     Tree t3 = new Tree(new Node(3),t4,t5),
12         t6 = new Tree(new Node(6),t7),
13         t2 = new Tree(new Node(2), t3, t6),
14         t1 = new Tree(new Node(1), t2,t10);
15     boolean repOk = t1.repOK();
16     if (repOk)
17         System.out.println(t1);
18     System.out.println("valid: " + repOk);
19     System.out.println("Some sub-trees:\n");
20     System.out.printf("%s%n%n",t4);
21     System.out.printf("%s%n%n",t6);
22     System.out.printf("%s%n%n",t10);
23 }
24 }
```

Top-Down Design

Listing 10.7 shows the Java implementation of the top-down class Tree. To conserve space, we only show the attributes, the constructor and the operation addNode. The code of other operations is the same as in the bottom-up Tree class (shown in Listing 10.5).

Listing 10.7: Java implementation of top-down Tree

```

1 import java.util.Vector;
2 import chap10.tree.Edge;
3 import chap10.tree.Node;
4 import utils.DomainConstraint;
5 /**
6  * @overview ...
7  *
8  * @attributes
9  *   root   Node
10 *   nodes  Set<Node>  Vector<Node>
11 *   edges  Set<Edge>  Vector<Edge>
12 *
13 * @object ...
14 *
```

```

15 * @abstract_properties ...
16 *
17 * @author dmle
18 */
19 public class Tree {
20     @DomainConstraint(type="Node",mutable=false,optional=false)
21     private Node root;
22     @DomainConstraint(type="Vector",mutable=true,optional=false)
23     private Vector<Node> nodes;
24     @DomainConstraint(type="Vector",mutable=true,optional=true)
25     private Vector<Edge> edges;
26
27     // constructors
28     /**
29     * @requires r != null
30     * @effects initialise this as <r,{r},{}>
31     */
32     public Tree(Node r) {
33         // single-node tree
34         this.root = r;
35         nodes = new Vector<>();
36         nodes.add(r);
37     }
38
39     /**
40     * @requires <tt>n != null \wedge parent != null</tt>
41     * @effects
42     *   if <tt>parent</tt> is in <tt>nodes</tt>
43     *   add <tt>n</tt> as a child of <tt>parent</tt>, i.e. <tt>edge(parent,n)</tt>
44     *   return true
45     * else
46     *   return false
47     */
48     public boolean addNode(Node parent, Node n) {
49         // check that parent is in nodes
50         boolean found = false;
51         for (Node node : nodes) {
52             if (node.equals(parent)) {
53                 found = true;
54                 break;
55             }
56         }

```

```

57     if (!found) {
58         return false;
59     }
60     // add node
61     nodes.add(n);
62
63     // create an edge <parent,n>
64     if (edges == null)
65         edges = new Vector<>();
66     Edge e = new Edge(parent, n);
67     edges.add(e);
68     return true;
69 }
70
71 // other methods (same as the bottom-up Tree class)
72 } // end Tree

```

The program class that uses the top-down Tree class is `chap10.test.tree.TopDownTreeTest` in the attached source code. Listing 10.8 shows the code of this class.

Listing 10.8: An example program class for the top-down Tree

```

1 import chap10.tree.Node;
2 import chap10.tree.topdown.Tree;
3 public class TopDownTreeTest {
4     public static void main(String[] args) {
5         Node n1 = new Node(1), n2 = new Node(2),
6             n3 = new Node(3), n4 = new Node(4),
7             n5 = new Node(5), n6 = new Node(6),
8             n7 = new Node(7), n8 = new Node(8),
9             n9 = new Node(9), n10 = new Node(10),
10            n11 = new Node(11), n12 = new Node(12);
11
12        Tree t = new Tree(n1);
13        System.out.printf("tree(1): %n%s%n",t);
14        boolean aok = false;
15        aok = t.addNode(n1,n2);
16        System.out.printf("addNode(1,2): %b%n",aok);
17        t.addNode(n2,n3); t.addNode(n2,n6);
18        System.out.printf("after adding 2, 3, 6: %n%s%n",t);
19        t.addNode(n1,n10); t.addNode(n10,n11); t.addNode(n10,n12);
20        t.addNode(n3,n4); t.addNode(n3,n5);
21        t.addNode(n6,n7); t.addNode(n7,n8); t.addNode(n7,n9);
22        boolean repOk = t.repOK();
23
24    if (repOk)

```

```

23     System.out.printf("complete tree: %n%s%n%n",t);
24     System.out.println("valid: " + rep0k);
25 }
26 }
```

10.4 List

10.4.1 Abstract Concept

List is basically a finite sequence of elements. List is empty if the number of elements is zero. Notationwise, a list of elements a_1, \dots, a_n is written as (a_1, a_2, \dots, a_n) . The empty list is written as () or by the symbol ϵ .

The following sequences are examples of list:

- (2, 3, 5, 7, 11, 13, 17, 19)
- (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
- a line of text: a sequence of characters
- a document: a sequence of lines
- a point in \mathbb{R}^n : a tuple consisting of n coordinates

List-Related Terms

The common list-related terms are: length, head, tail, sublist and subsequence [1].

Definition 10.14. Length

The length of a list is number of occurrences of its elements.



For example, the list $L = (2, 3, 5, 7, 11, 13, 17, 19)$ has length = 8.

Definition 10.15. Sublist

A sublist of a list is a list that contains zero or more consecutive elements.



Definition 10.16. Subsequence

A subsequence of a list is a sequence of 0 or more (not necessarily) consecutive elements.



For example, (2,3,5) is a sublist and (2,5,11,13,19) is a subsequence of the list L above.

Definition 10.17. Head

The head of a list is the first element.



Definition 10.18. Tail

The tail of a list is a sublist containing all but the first element.



For example, 2 is the head and (3, 5, 7, 11, 13, 17, 19) is the tail of the list L above.

Recursive Definition

List can be defined recursively in terms of head and tail [9].

Definition 10.19. List

Basis.

$()$ is a list.

Induction.

$\forall x, \forall L \in \text{List}. (x, L.\text{elements})$ is a list.

($L.\text{elements}$ is the sequence of elements of L)



Abstract Properties

List's properties are relatively straight-forward and are captured in Definition 10.20.

Definition 10.20. List properties

List has the following properties:

1. list is a sequence of elements
2. a non-empty list must have a head, tail may be empty



10.4.2 Operations

The following essential list operations are adapted from [1].

Operation insert

To insert an element an element into an arbitrary position in the list. A special case is to **push** an element x into the first position of a list (a_1, \dots, a_n) to become its new head. This creates a new list (x, a_1, \dots, a_n) .

Operation push

To insert an element at the front of the list (pushing the existing elements to the right).

Operation add

To add an element to the end of the list.

Operation remove

To remove an occurrence of an element from a list. If element is not in the list, the operation should not change the list. Otherwise, only the first occurrence of the element is removed.

Operation lookup

To return true or false depending on whether or not an element is in the list.

Operation concatenate

To form a new list from concatenating two lists. The new list begins with elements of the first list and continues with the elements the second list. Note that the empty list is the identity for concatenation. That is, for any list L , we have $\epsilon L = L\epsilon = L$.

Operation head

To return the head of the list.

Operation tail

To return the tail of the list.

Operation get

To return an element at a given position of the list. This operation is called `retrieve` in [1].

Operation size

To return the length of the list. This operation is called `length` in [1].

Operation isEmpty

To return true or false depending on whether or not the list is empty.

10.4.3 Design Approach

This section discusses a number of design questions that are specific to list.

How to represent the elements?

The most important design decision for list is which concrete type to use to represent its elements. In general, it is observed that list designs in the literature fall into one of two types: *recursive* and *non-recursive*. A design based on linked list or tree is a recursive design as it conforms to the recursive Definition 10.19. A design based on array, however, is non-recursive as it does not follow this recursive definition.

In the spirit of recursion, this section will focus on **two recursive designs**: one is called **recursive list** and the other is **linked list**. Recursive list is named as such because it is considered a pure recursive design, which precisely realises the recursive definition.

How to realise the recursive definition?

The recursive definition is realised as follows:

- Basis clause is realised by the default constructor
- Inductive clause is realised by two operations:
 - one other constructor, and
 - operation push

10.4.4 ‘Pure’ Recursive List

Figure 10.11 shows the ‘pure’ recursive and generic class design of List. For brevity, we will refer to this design simply as the recursive design. This design directly transforms the recursive definition into a recursive class design. However, unlike class Tree, which does not have any attributes of the same type, class List has an attribute name tail whose type is also List. This attribute keeps the reference to the list’s tail.

It may appear at first that attribute tail does not exactly match the inductive rule, which requires that only the elements of the existing list (not the list itself) are used to construct the new list. However, a closer look reveals that it logically is the same. The fact that the elements of a list becomes part of another list means that the list itself also belongs to that list. Further in object-oriented design, the fact that tail references a List object is an internal design detail, and is hidden from code that uses the class List. As far as the using code is concerned, List still represents a sequence of elements.

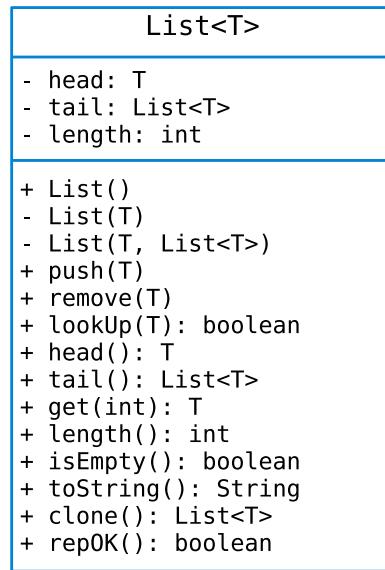


Figure 10.11: The ‘pure’ recursive design of `List`.

Derived attribute

Because operation `size()` involves recursively counting the number of occurrences of all the tail sub-lists of a list, invoking it many times may affect the overall performance of the program. The derived attribute pattern is applied to overcome this. This involves creating a derived attribute named `sz` and using it to record the current number of elements of the list. To keep this attribute current, it is updated when a new element is inserted to or removed from the list.

Note that since derived attributes are not essential to the abstract concept, they need not be listed in the `@attributes` section of the header specification.

Constructors

Class `List` has three constructors: one public and two private. The public constructor implements the basis rule of the recursive definition. The one-argument private constructor together with the operation `push` implements the inductive rule of the definition. The two-argument private constructor is used by the operation `clone` to copy the `tail` before returning it.

Operation size

For performance reasons, this operation returns the value of the derived attribute `sz`, instead of computing it using recursion.

Operation tail

This operation does not return `tail` directly. Instead it makes a copy of the `List` object that is referenced by this attribute and returns a reference to it. This helps protect this attribute from modifications outside of class.

Operation clone

This operation returns a copy of the current list object. It is used by the `tail()` operation mentioned above. The copy is another `List` object that has the same content as the current object.

Design Specification

Listing 10.9 shows the design specification of the recursive design of `List`. Note the followings:

- the recursive definition is inserted in the `@object` section
- attribute `head` has the same type as the `List`'s type variable `T`
- attribute `tail` has the type `List<T>`
- the `@abstract_properties` section states the two `List`'s properties in terms of the two attributes `head`, `tail`

Listing 10.9: Recursive design specification of `List`

```

1 import utils.AttrRef;
2 import utils.DOpt;
3 import utils.DomainConstraint;
4 import utils.OptType;
5 import utils.collections.Collection;
6 /**
7 * @overview Represents a (possibly empty) list of elements of arbitrary
8 *          types.
9 *
10 * @attributes
11 *      head    T
12 *      tail    List<T>
13 *
14 *      @object
15 *      A typical list is c = (x1, ..., xn), where head(x1) and tail(x2, ..., xn)
16 *          are elements.
17 *
18 *      <p>Recursive definition:

```

```

17 * Basis
18 * () is a list
19 * Induction
20 * for all object x and for all list L, a new list is formed by inserting
21 * x
22 * into the first position of L to become a head, thereby making the
23 * existing elements of L become the tail of the new list.
24 *
25 * @abstract_properties
26 * mutable(head)=false /\ optional(head)=true /\
27 * mutable(tail)=true /\ optional(tail)=true /\
28 * tail = (x1,...,xk) is a sequence /\ (head,x1,...,xk) is a sequence /\
29 * tail is not empty -> head is initialised
30 * @author dmle
31 */
32 public class List<T> implements Collection<T> {
33     @DomainConstraint(type="T",mutable=false,optional=true)
34     private T head;
35     @DomainConstraint(type="List<T>",mutable=true,optional=true)
36     private List<T> tail;
37     // derived attribute
38     @DomainConstraint(type="Integer",mutable=false,optional=true)
39     private int sz;
40 
41     /**
42      * @effects initialises this to be ()
43      */
44     public List()
45     /**
46      * This constructor together with operation <tt>push</tt> implement the
47      * inductive rule.
48      */
49     private List(T x)
50     /**
51      * This constructor is used to clone a list.
52      */
53     /**
54      * @requires h != null
55      * @effects initialise this to be (h,x1,...,xn) where t = (x1,...,xn)
56      */
57     private List(T h, List<T> t)

```

```

58  /**
59   * This operation implements the inductive rule.
60   *
61   * @requires x != null
62   * @modifies this
63   * @effects
64   * if this is empty
65   *   head = x
66   * else
67   *   inserts x into the first position of this (pushing
68   *   the element at the current position to the right)
69   */
70  public void push(T x)
71  /**
72   * This operation implements the inductive rule.
73   *
74   * @requires x != null
75   * @modifies this
76   * @effects
77   * if this is empty
78   *   head = x
79   * else
80   *   adds x to the end of this
81   */
82  @DOpt(type=OptType.MutatorAdd)
83  public void add(T x)
84  /**
85   * @requires x != null
86   * @modifies this
87   *
88   * @effects
89   * if this is empty or x is not in this
90   * do nothing
91   * else
92   * remove the first occurrence of x from this
93   *
94   * i.e. (recursive)
95   * ...
96   */
97  @DOpt(type=OptType.MutatorRemove)
98  public void remove(T x)
99  /**
100  * @requires x != null

```

```

101   * @effects
102   *   if x is in this
103   *     return true
104   *   else
105   *     return false
106   */
107 @DOpt(type=OptType.ObserverContains)
108 public boolean lookUp(T x)
109 /**
110   * @effects
111   *   return head
112   */
113 @DOpt(type=OptType.Observer) @AttrRef("head")
114 public T head()
115 /**
116   * @effects
117   *   if this is empty or a single-element list
118   *     return null
119   *   else
120   *     return tail as a new list
121   */
122 @DOpt(type=OptType.Observer) @AttrRef("tail")
123 public List<T> tail()
124 /**
125   * @requires index >= 0
126   * @effects
127   *   if this is empty or index is out of range
128   *     return null
129   *   else
130   *     return the element at the position index
131   */
132 @DOpt(type=OptType.Observer)
133 public T get(int index)
134 /**
135   * @effects
136   *   return the number of occurrences of the elements of this
137   */
138 @DOpt(type=OptType.ObserverSize)
139 public int size()
140 /**
141   * @effects
142   *   if this is empty
143   *     return true

```

```

144     * else
145     * return false
146     */
147     @DOpt(type=OptType.Observer)
148     public boolean isEmpty()
149     @Override
150     public String toString()
151     /**
152     * @effects
153     * return a copy of this
154     */
155     @Override
156     public List<T> clone()
157     /**
158     * @effects
159     * if this satisfies abstract properties
160     * return true
161     * else
162     * return false
163     */
164     public boolean repOK()
165 } /** end {@link List} */

```

10.4.5 Linked List

Another design, which is suggested in [1] (but not specified as an object oriented solution), is to encapsulate both head and tail using a cell. Figure 10.12 illustrates this design. Given a list $L = (a_1, a_2, \dots, a_n)$, the list object variable for L points to the first cell, the tail of this cell points to the second cell, and so on. The tail of the last cell is null, since the list terminates here. The head of each cell contains the value of each element.

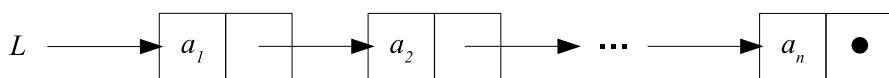


Figure 10.12: Cell-based design for list (Adapted from [1]).

To build this design, we create a generic class `LinkedList<T>` and introduce an inner class, named `Cell<T>`. Figure 10.13 shows the class diagram of this design. Since class `Cell` now has `head` and `tail`, it becomes the recursive class. Class `LinkedList` simply contains a reference to `Cell` object of the list's first element.

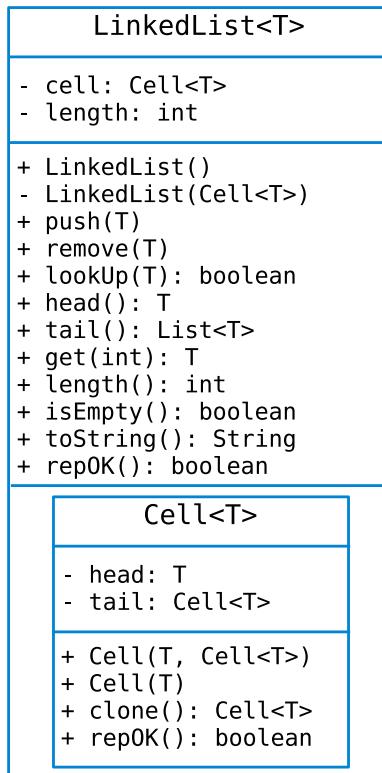


Figure 10.13: The linked ‘recursive’ design of List.

Design specification

Listing 10.10 shows the partial specification of class `LinkedList`. In this, we highlight the differences between this specification and the specification of the recursive design in Section 10.4.4. The operational specification is omitted as it is the same as the specification of that design.

Note the followings about the design specification:

- @object section is defined in terms of `cell.head` and `cell.tail`
- the abstract properties regarding head and tail is moved to class `Cell`
- class `Cell` is a private inner class
- operation `clone` is moved to class `Cell` because `tail` is defined there

Listing 10.10: Design specification of `LinkedList`

```

1 import utils.AttrRef;
2 import utils.DOpt;
3 import utils.DomainConstraint;
4 import utils.OptType;
5 import utils.collections.Collection;
6 /**
7 * @overview Represents a (possibly empty) list of elements of arbitrary
8 * types.
9 */

```

```

9  * @attributes
10 *   cell    Cell<T>
11 *
12 * @object
13 *   A typical list is c = (x1,...,xn), where cell.head(x1) and cell.tail(x2
14 *           ,...,xn) are elements.
15 *   <p>Recursive definition:
16 *     Basis
17 *       () is a list
18 *     Induction
19 *       for all object x and for all list L, a new list is formed by inserting
20 *           x
21 *           into the first position of L to become a head, thereby making the
22 *           existing elements of L become the tail of the new list.
23 *
24 * @abstract_properties
25 *   mutable(cell)=true /\ optional(cell)=true /\ P_Cell
26 *
27 * @author dmle
28 */
29 public class LinkedList<T> implements Collection<T> {
30   @DomainConstraint(type="Cell<T>",mutable=true,optional=true)
31   private Cell<T> cell;
32   // derived attribute
33   @DomainConstraint(type="Integer",mutable=false,optional=true)
34   private int sz;
35   // constructors
36 /**
37   * @effects initialises this to be ()
38 */
39 public LinkedList() {
40   // empty list
41 }
42 /**
43   * @requires n != null
44   * @effects initialises this to be a list (n.head)
45 */
46 private LinkedList(Cell<T> n) {
47   cell = n;
48 }
49

```

```

50  // ... other operations (omitted)...
51
52 /**
53 * @overview Represents a linked cell that contains a head value of the
54 *           current list and a nullable
55 *           reference pointer to another cell representing the tail of the
56 *           current list.
57 *
58 * @attributes
59 *   head Object
60 *   tail Cell
61 *
62 * @object
63 *   A typical cell is <v,r>, where head(v) and tail(r).
64 *
65 * @abstract_properties
66 *   mutable(head)=false /\ optional(head)=false /\
67 *   mutable(tail)=false /\ optional(tail)=true
68 *
69 * @author dmle
70 */
71 private class Cell<T> {
72     private T head;
73     private Cell<T> tail;
74
75     // constructors
76 /**
77 * @requires v != null
78 * @effects
79 *   initialise this to be <v,r>
80 */
81     public Cell(T v, Cell<T> r)
82     /**
83 * @requires v != null
84 * @effects
85 *   initialise this to be <v,null>
86 */
87     public Cell(T v)
88     /**
89 * @effects
90 *   sets this.{@link #tail} = tail
91 */
92     public void setTail(Cell<T> tail)

```

```

91     /**
92      * @effects
93      *   return {@link #tail}
94      */
95     public Cell<T> getTail()
96     /**
97      * @effects
98      *   return a deep copy of this
99      */
100    @Override
101    public Cell<T> clone()
102    /**
103     * @effects
104     *   if this satisfies abstract properties
105     *   return true
106     *   else
107     *   return false
108     */
109    public boolean repOK()
110  } /** end {@link LinkedList.Cell} */
111 } /** end {@link LinkedList} */

```

10.4.6 Implementation

10.4.6.1 ‘Pure’ Recursive List

Listing 10.11 shows the Java implementation of the recursive List class.

Listing 10.11: Java implementation of recursive List

```

1 import utils.AttrRef;
2 import utils.DOpt;
3 import utils.DomainConstraint;
4 import utils.OptType;
5 import utils.collections.Collection;
6 /**
7  * @overview ...
8  * @attributes ...
9  * @object ...
10 * @abstract_properties ...
11 */
12 public class List<T> implements Collection<T> {
13   @DomainConstraint(type="T",mutable=false,optional=true)
14   private T head;

```

```

15  @DomainConstraint(type="List<T>",mutable=true,optional=true)
16  private List<T> tail;
17  // derived attribute
18  @DomainConstraint(type="Integer",mutable=false,optional=true)
19  private int sz;
20
21 /**
22 * @effects initialises this to be ()
23 */
24 public List() {
25     // empty list
26 }
27
28 /**
29 * This constructor together with operation <tt>push</tt> implement the
30     inductive rule.
31 *
32 * @requires x != null
33 * @effects initialises this to be (x)
34 */
35 private List(T x) {
36     // make x the head
37     head = x;
38
39     // update length
40     sz++;
41 }
42 /**
43 * This constructor is used to clone a list.
44 *
45 * @requires h != null
46 * @effects initialise this to be (h,x1,...,xn) where t = (x1,...,xn)
47 */
48 private List(T h, List<T> t) {
49     head = h;
50     tail = t;
51 }
52
53 /**
54 * This operation implements the inductive rule.
55 *
56 * @requires x != null

```

```

57  * @modifies this
58  * @effects
59  * if this is empty
60  *   head = x
61  * else
62  *   inserts x into the first position of this (pushing
63  *   the element at the current position to the right)
64  */
65 public void push(T x) {
66     if (head == null) {
67         // make x the head
68         head = x;
69     } else {
70         // push head into tail
71         if (tail == null) {
72             tail = new List<>(head);
73         } else {
74             tail.push(head);
75         }
76
77         // make x the head
78         head = x;
79     }
80
81     // update length
82     sz++;
83 }
84
85 /**
86 * This operation implements the inductive rule.
87 *
88 * @requires x != null
89 * @modifies this
90 * @effects
91 * if this is empty
92 *   head = x
93 * else
94 *   adds x to the end of this
95 */
96 @DOpt(type=OptType.MutatorAdd)
97 public void add(T x) {
98     if (head == null) {
99         // make x the head

```

```

100     head = x;
101 } else {
102     // keep head the same, add x to tail
103     if (tail == null) {
104         tail = new List<x>(x);
105     } else {
106         tail.add(x);
107     }
108 }
109
110     // update length
111     sz++;
112 }
113
114 /**
115 * @requires x != null
116 * @modifies this
117 *
118 * @effects
119 *   if this is empty or x is not in this
120 *   do nothing
121 *   else
122 *       remove the first occurrence of x from this
123 *
124 *   i.e. (recursive)
125 *   ...
126 */
127 @DOpt(type=OptType.MutatorRemove)
128 public void remove(T x) {
129     if (head == null)
130         return;
131
132
133     if (head.equals(x)) {
134         // x is head
135         if (tail != null) {
136             // has tail: make tail.head the head
137             head = tail.head;
138             tail.remove(head);
139         } else {
140             // no tail: nullify head
141             head = null;
142         }

```

```

143     } else if (tail != null) {
144         // x is not head /\ has tail: recursively remove x in tail
145         tail.remove(x);
146     }
147
148     // cut tail if empty
149     if (tail != null && tail.isEmpty()) {
150         tail = null;
151     }
152
153     // update length
154     sz--;
155 }
156
157 /**
158 * @requires x != null
159 * @effects
160 *   if x is in this
161 *   return true
162 *   else
163 *   return false
164 */
165 @DOpt(type=OptType.ObserverContains)
166 public boolean lookUp(T x) {
167     if (head == null)
168         return false;
169
170     if (head.equals(x)) {
171         return true;
172     } else if (tail != null) {
173         return tail.lookUp(x);
174     } else {
175         return false;
176     }
177 }
178
179 /**
180 * @effects
181 *   return head
182 */
183 @DOpt(type=OptType.Observer) @AttrRef("head")
184 public T head() {
185     return head;

```

```

186     }
187
188     /**
189      * @effects
190      * if this is empty or a single-element list
191      *   return null
192      * else
193      *   return tail as a new list
194      */
195     @DOpt(type=OptType.Observer) @AttrRef("tail")
196     public List<T> tail() {
197         if (tail == null) {
198             return null;
199         } else {
200             return tail.clone();
201         }
202     }
203
204     /**
205      * @requires index >= 0
206      * @effects
207      * if this is empty or index is out of range
208      *   return null
209      * else
210      *   return the element at the position index
211      */
212     @DOpt(type=OptType.Observer)
213     public T get(int index) {
214         T h = head;
215         List<T> t = tail;
216
217         int count = 0;
218         while (count < index) {
219             if (t == null) { // out of range
220                 return null;
221             } else {
222                 h = t.head;
223                 t = t.tail;
224                 count++;
225             }
226         }
227
228         return h;

```

```

229     }
230
231     /**
232      * @effects
233      *   return the number of occurrences of the elements of this
234      */
235     @DOpt(type=OptType.ObserverSize)
236     public int size() {
237         return sz;
238     }
239
240     /**
241      * @effects
242      *   if this is empty
243      *   return true
244      *   else
245      *   return false
246      */
247     @DOpt(type=OptType.Observer)
248     public boolean isEmpty() {
249         if (head == null) // empty
250             return true;
251         else
252             return false;
253     }
254
255     @Override
256     public String toString() {
257         StringBuffer sb = new StringBuffer();
258         if (head == null) { // empty
259             sb.append("()");
260         } else {
261             sb.append("(");
262             T h = head;
263             List<T> t = tail;
264             sb.append(h);
265             while (t != null) {
266                 sb.append(",");
267                 sb.append(t.head);
268                 t = t.tail;
269             }
270             sb.append(")");
271     }

```

```

272
273     return sb.toString();
274 }
275
276 /**
277 * @effects
278 *   return a copy of this
279 */
280 @Override
281 public List<T> clone() {
282     if (head == null)
283         return null;
284
285     T h = head;
286     List<T> t;
287     if (tail != null) {
288         t = tail.clone();
289     } else {
290         t = null;
291     }
292
293     return new List<>(h,t);
294 }
295
296 /**
297 * @effects
298 *   if this is valid
299 *   return true
300 *   else
301 *   return false
302 */
303 public boolean repOK() {
304     if (tail != null && head == null)
305         return false;
306
307     return true;
308 }
309 } /** end {@link List} */

```

The program class that uses the List class is chap10.test.list.ListTest in the attached source code. Listing 10.12 shows the code of this class.

Listing 10.12: An example program class for the recursive List

```
1 import chap10.list.List;
```

```

2 public class ListTest {
3     public static void main(String[] args) {
4         // test data
5         Object[] objs = {"hello world!", 2, 0, 1, 3};
6         Object[] nonMembers = {null, -1, 0, "", 4};
7
8         List l = new List();
9         System.out.printf("initial: %s%n", l);
10        Object a = 1;
11        int index = 1;
12        System.out.printf("lookUp(%s) -> %b%n", a,l.lookUp(a));
13        System.out.printf("get(%d) -> %s%n", index, l.get(index));
14        System.out.printf("size() -> %d%n", l.size());
15        System.out.printf("head() -> %s%n", l.head());
16        System.out.printf("tail() -> %s%n", l.tail());
17        System.out.printf("isEmpty() -> %b%n", l.isEmpty());
18
19        // push
20        System.out.println("push:\n=====");
21        for (Object o : objs) {
22            l.push(o);
23            System.out.printf("push(%s) -> %s%n", o,l);
24        }
25
26        System.out.printf("after: %s%n", l);
27        System.out.printf("size() -> %d%n", l.size());
28        System.out.printf("head() -> %s%n", l.head());
29        System.out.printf("tail() -> %s%n", l.tail());
30        System.out.printf("isEmpty() -> %b%n", l.isEmpty());
31
32        // add
33        System.out.println("add:\n=====");
34        for (Object o : objs) {
35            l.add(o);
36            System.out.printf("add(%s) -> %s%n", o,l);
37        }
38
39        System.out.printf("after: %s%n", l);
40        System.out.printf("size() -> %d%n", l.size());
41        System.out.printf("head() -> %s%n", l.head());
42        System.out.printf("tail() -> %s%n", l.tail());
43        System.out.printf("isEmpty() -> %b%n", l.isEmpty());
44

```

```
45 // look up
46 System.out.println("lookup:\n=====");
47 boolean tf;
48 for (Object o : objs) {
49     tf = l.lookUp(o);
50     System.out.printf("lookUp(%s) -> %b\n", o, tf);
51 }
52
53 for (Object o : nonMembers) {
54     tf = l.lookUp(o);
55     System.out.printf("lookUp(%s) -> %b\n", o, tf);
56 }
57
58 // get
59 System.out.println("get:\n=====");
60 int size = l.size();
61 for (int i = 0; i < size; i++) {
62     Object o = l.get(i);
63     System.out.printf("get(%d) -> %s\n", i, o);
64 }
65
66 index = 5;
67 a = l.get(index);
68 System.out.printf("get(%d) -> %s\n", index, a);
69
70 // remove
71 System.out.println("remove:\n=====");
72 for (Object o : objs) {
73     l.remove(o);
74     System.out.printf("remove(%s) -> %s\n", o, l);
75 }
76 System.out.printf("after: %s\n", l);
77
78 System.out.printf("size() -> %d\n", l.size());
79 System.out.printf("head() -> %s\n", l.head());
80 System.out.printf("tail() -> %s\n", l.tail());
81 System.out.printf("isEmpty() -> %b\n", l.isEmpty());
82 }
83 }
```

10.4.6.2 Linked List

Listing 10.13 shows the Java implementation of the LinkedList class.

Listing 10.13: Java implementation of LinkedList

```

1 import utils.AttrRef;
2 import utils.DOpt;
3 import utils.DomainConstraint;
4 import utils.OptType;
5 import utils.collections.Collection;
6 /**
7  * @overview ...
8  * @attributes ...
9  * @object ...
10 * @abstract_properties ...
11 */
12 public class LinkedList<T> implements Collection<T> {
13     @DomainConstraint(type="Cell<T>",mutable=true,optional=true)
14     private Cell<T> cell;
15
16     // derived attribute
17     @DomainConstraint(type="Integer",mutable=false,optional=true)
18     private int sz;
19
20     // constructors
21     /**
22      * @effects initialises this to be ()
23      */
24     public LinkedList() {
25         // empty list
26     }
27
28     /**
29      * @requires n != null
30      * @effects initialises this to be a list (n.head)
31      */
32     private LinkedList(Cell<T> n) {
33         cell = n;
34     }
35
36     /**
37      * @requires x != null
38      * @effects initialises this to be (x)
39      */

```

```

40     public LinkedList(T x) {
41         cell = new Cell<x>(x);
42         sz++;
43     }
44
45     /**
46      * @requires x != null
47      * @modifies this
48      * @effects
49      * if this is empty
50      * initialise cell such that cell.head = x
51      * else
52      * inserts x into the first position of this (pushing
53      * the element at the current position to the right)
54      */
55     public void push(T x) {
56         if (cell == null) {
57             cell = new Cell<x>(x);
58         } else {
59             // push x
60             cell = new Cell<x>(x, cell);
61         }
62         sz++;
63     }
64
65     /**
66      * @requires x != null
67      * @modifies this
68      * @effects
69      * if this is empty
70      * initialise cell such that cell.head = x
71      * else
72      * adds x to the end of this
73      */
74     @DOpt(type=OptType.MutatorAdd)
75     public void add(T x) {
76         if (cell == null) {
77             cell = new Cell<x>(x);
78         } else {
79             // add x to the tail of the last cell
80             Cell<T> tail = cell.getTail(), last = null;
81             while (tail != null) {
82                 last = tail;

```

```

83         tail = last.getTail();
84     }
85
86     if (last == null) { // cell is the last one
87         cell.setTail(new Cell(x));
88     } else {
89         last.setTail(new Cell(x));
90     }
91 }
92 sz++;
93 }
94
95 /**
96 * @requires x != null
97 * @modifies this
98 *
99 * @effects
100 *   if this is empty or x is not in this
101 *   do nothing
102 *   else
103 *   remove the first occurrence of x from this
104 */
105 @DOpt(type=OptType.MutatorRemove)
106 public void remove(T x) {
107     boolean found = false;
108     Cell<T> n = cell;
109     Cell<T> prev = null;
110     while (!found && n != null) {
111         Cell<T> tail = n.tail;
112         if (n.head.equals(x)) { // compare using equals
113             if (prev != null)
114                 prev.tail = tail;
115             else
116                 cell = tail;
117             found = true;
118             sz--;
119         } else {
120             prev = n;
121             n = tail;
122         }
123     }
124 }
125

```

```

126 /**
127 * @requires x != null
128 * @effects
129 *   if x is in this
130 *     return true
131 *   else
132 *     return false
133 */
134 @DOpt(type=OptType.ObserverContains)
135 public boolean lookUp(T x) {
136     boolean found = false;
137     Cell<T> n = cell;
138     while (!found && n != null) {
139         if (n.head.equals(x)) { // compare using equals
140             found = true;
141         }
142         n = n.tail;
143     }
144
145     return found;
146 }
147
148 /**
149 * @effects
150 *   return head
151 */
152 @DOpt(type=OptType.Observer) @AttrRef("head")
153 public T head() {
154     if (cell != null)
155         return cell.head;
156     else
157         return null;
158 }
159
160 /**
161 * @effects
162 *   if this is empty or a single-element list
163 *     return null
164 *   else
165 *     return tail as a new list
166 */
167 @DOpt(type=OptType.Observer) @AttrRef("tail")
168 public LinkedList<T> tail() {

```

```

169     if (cell != null) { /* this is not empty */
170         Cell<T> tail = cell.tail;
171         if (tail != null) {
172             // return as a new list
173             Cell<T> ctail = tail.clone();
174             LinkedList<T> l = new LinkedList<>(ctail);
175             return l;
176         } else {
177             return null;
178         }
179     } else {
180         return null;
181     }
182 }
183
184 /**
185 * @requires index >= 0
186 * @effects
187 *   if this is empty or index is out of range
188 *   return null
189 *   else
190 *       return the element at the position index
191 */
192 @DOpt(type=OptType.Observer)
193 public T get(int index) {
194     if (cell == null) {
195         return null;
196     } else {
197         int count = 0;
198         Cell<T> n = cell;
199         while (n != null) {
200             if (count == index) {
201                 return n.head;
202             } else {
203                 count++;
204                 n = n.tail;
205             }
206         }
207
208         return null;
209     }
210 }
211

```

```

212 /**
213 * @effects
214 * returns the number of occurrences of the elements of this
215 */
216 @DOpt(type=OptType.ObserverSize)
217 public int size() {
218     return sz;
219 }
220
221 /**
222 * @effects
223 * if this is empty
224 * return true
225 * else
226 * return false
227 */
228 @DOpt(type=OptType.Observer)
229 public boolean isEmpty() {
230     if (cell == null) // empty
231         return true;
232     else
233         return false;
234 }
235
236
237 @Override
238 public String toString() {
239     StringBuffer sb = new StringBuffer();
240     if (cell == null) { // empty
241         sb.append("()");
242     } else {
243         sb.append("(");
244         Cell<T> n = cell;
245         while (n != null) {
246             sb.append(n.head);
247             n = n.tail;
248             if (n != null)
249                 sb.append(",");
250         }
251         sb.append(")");
252     }
253
254     return sb.toString();

```

```

255     }
256
257     /**
258      * @overview Represents a linked cell that contains a head value of the
259      *           current list and a nullable
260      *           reference pointer to another cell representing the tail of the
261      *           current list.
262      *
263      * @attributes
264      *   head   Object
265      *   tail   Cell
266      *
267      * @object
268      *   A typical cell is <v,r>, where head(v) and tail(r).
269      *
270      * @abstract_properties
271      *   mutable(head)=false /\ optional(head)=false /\
272      *   mutable(tail)=false /\ optional(tail)=true
273      *
274      * @rep_invariant
275      *   head != null
276      *
277      * @author dmle
278      */
279
280     private class Cell<T> {
281
282         private T head;
283         private Cell<T> tail;
284
285         // constructors
286         /**
287          * @requires v != null
288          * @effects
289          *   initialise this to be <v,r>
290          */
291
292         public Cell(T v, Cell<T> r) {
293
294             head = v;
295             tail = r;
296         }
297
298         /**
299          * @requires v != null
300          * @effects
301          *   initialise this to be <v,null>
302
303     }

```

```

296     */
297     public Cell(T v) {
298         this(v, null);
299     }
300
301 /**
302 * @effects
303 *   sets this.{@link #tail} = tail
304 */
305 public void setTail(Cell<T> tail) {
306     this.tail = tail;
307 }
308
309 /**
310 * @effects
311 *   return {@link #tail}
312 */
313 public Cell<T> getTail() {
314     return tail;
315 }
316
317 /**
318 * @effects
319 *   return a deep copy of this
320 */
321 @Override
322 public Cell<T> clone() {
323     // deep clone
324     T v = head;
325
326     // if ref is assigned, clone it as well
327     Cell<T> r = (tail != null) ? tail.clone() : null;
328
329     Cell<T> n = new Cell<>(v,r);
330     return n;
331 }
332
333 /**
334 * @effects
335 *   if this satisfies abstract properties
336 *   return true
337 *   else
338 *   return false

```

```

339     */
340     public boolean repOK() {
341         if (head == null)
342             return false;
343
344         return true;
345     }
346 } /** end {@link LinkedList.Cell} */
347 } /** end {@link LinkedList} */

```

The program class that uses the `LinkedList` class is `chap10.test.list.LinkedListTest` in the attached source code. Listing 10.14 shows the code of this class.

Listing 10.14: An example program class for `LinkedList`

```

1 import chap10.list.LinkedList;
2
3 public class LinkedListTest {
4     public static void main(String[] args) {
5         // test data
6         Object[] objs = {"hello world!", 2, 0, 1, 3};
7         Object[] nonMembers = {null, -1, 0, "", 4};
8
9         LinkedList l = new LinkedList();
10
11         System.out.printf("initial: %s%n", l);
12
13         Object a = 1;
14         int index = 1;
15         System.out.printf("lookUp(%s) -> %b%n", a, l.lookUp(a));
16         System.out.printf("get(%d) -> %s%n", index, l.get(index));
17         System.out.printf("size() -> %d%n", l.size());
18         System.out.printf("head() -> %s%n", l.head());
19         System.out.printf("tail() -> %s%n", l.tail());
20         System.out.printf("isEmpty() -> %b%n", l.isEmpty());
21
22         // insert
23         System.out.println("push:\n=====");
24         for (Object o : objs) {
25             l.push(o);
26             System.out.printf("push(%s) -> %s%n", o, l);
27         }
28
29         System.out.printf("after: %s%n", l);
30         System.out.printf("size() -> %d%n", l.size());

```

```

31     System.out.printf("head() -> %s%n", l.head());
32     System.out.printf("tail() -> %s%n", l.tail());
33     System.out.printf("isEmpty() -> %b%n", l.isEmpty());
34
35     // add
36     System.out.println("add:\n=====");
37     for (Object o : objs) {
38         l.add(o);
39         System.out.printf("add(%s) -> %s%n", o,l);
40     }
41
42     System.out.printf("after: %s%n", l);
43     System.out.printf("size() -> %d%n", l.size());
44     System.out.printf("head() -> %s%n", l.head());
45     System.out.printf("tail() -> %s%n", l.tail());
46     System.out.printf("isEmpty() -> %b%n", l.isEmpty());
47
48     // look up
49     System.out.println("lookup:\n=====");
50     boolean tf;
51     for (Object o : objs) {
52         tf = l.lookUp(o);
53         System.out.printf("lookUp(%s) -> %b%n", o,tf);
54     }
55
56     for (Object o : nonMembers) {
57         tf = l.lookUp(o);
58         System.out.printf("lookUp(%s) -> %b%n", o,tf);
59     }
60
61     // get
62     System.out.println("get:\n=====");
63     int length = l.size();
64     for (int i = 0; i < length; i++) {
65         Object o = l.get(i);
66         System.out.printf("get(%d) -> %s%n", i,o);
67     }
68
69     index = length+1;
70     a = l.get(index);
71     System.out.printf("get(%d) -> %s%n", index,a);
72
73     // remove

```

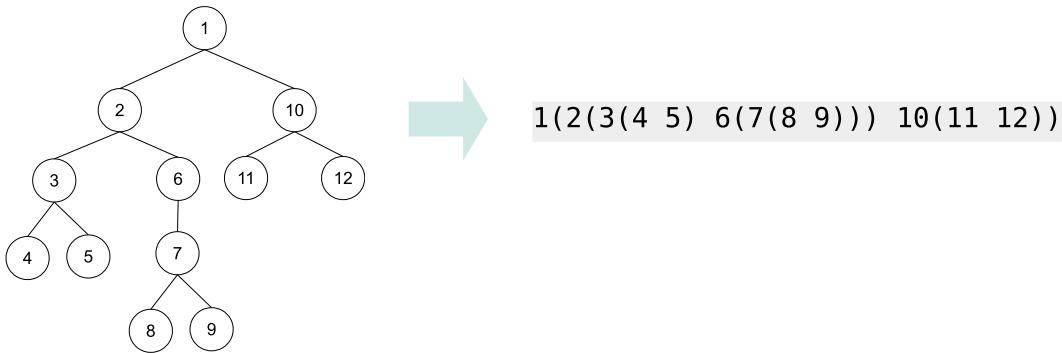
```
74     System.out.println("remove:\n=====");
75     for (Object o : objs) {
76         l.remove(o);
77         System.out.printf("remove(%s) -> %s%n", o, l);
78     }
79     System.out.printf("after: %s%n", l);
80
81     System.out.printf("size() -> %d%n", l.size());
82     System.out.printf("head() -> %s%n", l.head());
83     System.out.printf("tail() -> %s%n", l.tail());
84     System.out.printf("isEmpty() -> %b%n", l.isEmpty());
85 }
86 }
```

Chapter 10 Exercise

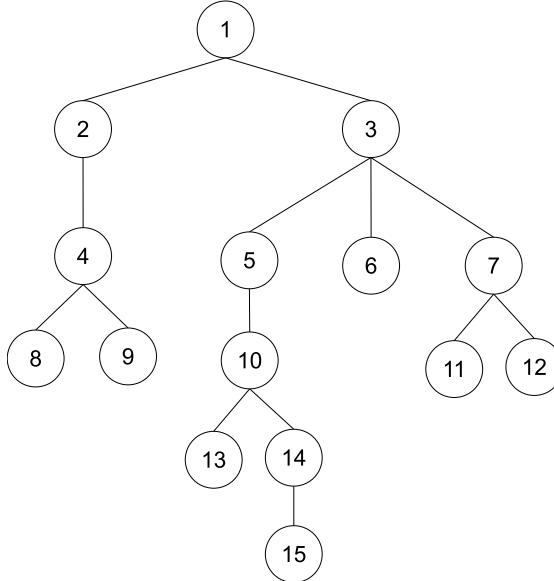
The exercises in this chapter help practise using and extending the design of list and tree. Some exercises are adopted from [1].

Tree.

- Design and implement another recursive version of the operation `Tree.toString` which produces a horizontal, textual representation of the tree, as shown in the illustration below:



- (Ex 5.2.1 [1]) For the tree below, identify the following elements:



- (a). root of the tree
- (b). leaves of the tree
- (c). interior nodes of the tree
- (d). siblings of node 6
- (e). subtree with root 5
- (f). ancestors of node 10
- (g). descendants of node 10
- (h). longest path in the tree
- (i). height of node 3
- (j). depth of node 13
- (k). height of the tree

- (Ex 5.2.2 [1]) Can a leaf in a tree ever have any (a) descendants? (b) proper descendants?
- (Ex 5.2.3 [1]) Prove that in a tree no leaf can be an ancestor of another leaf.
- (Ex 5.3.6 [1]) In a tree, a node c is the lowest common ancestor of nodes x and y if c is an ancestor of both x and y, and no proper descendant of c is an ancestor of x and y. Write a program that will find the lowest common ancestor of any pair of nodes in a given tree. What is a good design specification for trees in such a program?

6. (Ex 5.4.2 [1]) Write a recursive program to find the maximum label of the nodes of a tree. Assume that a node's label is the string representation of its state.

List.

7. (Ex 6.2.1 [1]) Answer the following questions about the list (2, 7, 1, 8, 2).
- (a). What is the head?
 - (b). What is the tail?
 - (c). What is the size?
 - (d). What are all the sublists?
 - (e). How many subsequences are there?
8. (Adapted from Ex 6.2.3 [1])): In a list of size $n \geq 0$, what are all the possible subsequences?
9. The code of recursive List class shows that operations `get` and `toString` follows an iterative not recursive implementation. How would you change these operations to follow the recursive implementation?
10. All the operations of class `LinkedList` followed the iterative implementation. Discuss the possibility of changing these operations to follow the recursive implementation.
11. Compare and contrast the implementations of the two operations `List.push` and `LinkedList.push`.
12. Design and implement a class named `DoubleLinkedList` [1] that represents linked list in which each cell maintains a reference to the previous cell.

10.A Node

The design and Java code of class Node.

Listing 10.15: Class Node

```

1 package chap10.tree;
2 import utils.DomainConstraint;
3 /**
4  * @overview
5  * Represents a labelled node.
6  * @attributes
7  * label T
8  * @abstract_properties
9  * mutable(label)=true /\ optional(label) = false
10 * @author Duc Minh Le (ducmlle)
11 */
12 public class Node<T> {
13     @DomainConstraint(mutable=true,optional=false)
14     private T label;
15
16     /**
17      * @requires label != null /\ label.length() > 0
18      * @effects
19      * initialise this as Node(label)
20      */
21     public Node(T label) {
22         this.label = label;
23     }
24
25     /**
26      * @effects
27      * return label
28      */
29     public T getLabel() {
30         return label;
31     }
32
33     /**
34      * @requires label != null /\ label.length() > 0
35      * @effects
36      * sets this.label = label
37      */
38     public void setLabel(T label) {

```

```

39     this.label = label;
40 }
41
42 @Override
43 public String toString() {
44     return toString(false);
45 }
46
47 public String toString(boolean full) {
48     if (full)
49         return "Node(\"+label+)";
50     else
51         return label.toString();
52 }
53
54 @Override
55 public boolean equals(Object o) {
56     if (o == null || !(o instanceof Node))
57         return false;
58
59     return ((Node)o).label.equals(label);
60 }
61 }

```

10.B Edge

The design and Java code of class Edge.

Listing 10.16: Class Edge

```

1 package chap10.tree;
2 import utils.DomainConstraint;
3 /**
4 * @overview
5 * Represents a binary, directed edge.
6 * @attributes
7 * src Node
8 * tgt Node
9 * @abstract_properties
10 * mutable(src)=true /\ optional(src) = false /\
11 * mutable(tgt)=true /\ optional(tgt) = false
12 * @author Duc Minh Le (ducml)
13 */

```

```

14 public class Edge {
15     @DomainConstraint(type="Node",mutable=true,optional=false)
16     private Node src;
17     @DomainConstraint(type="Node",mutable=true,optional=false)
18     private Node tgt;
19
20     /**
21      * @requires src != null \wedge tgt != null
22      * @effects
23      *   initialise this as Edge(src,tgt)
24      */
25     public Edge(Node src, Node tgt) {
26         this.src = src;
27         this.tgt = tgt;
28     }
29
30     /**
31      * @requires src != null
32      * @effects
33      *   sets this.src = src
34      */
35     public void setSrc(Node src) {
36         if (src != null) {
37             this.src = src;
38         }
39     }
40
41     /**
42      * @effects
43      *   return src
44      */
45     public Node getSrc() {
46         return src;
47     }
48
49     /**
50      * @requires tgt != null
51      * @effects
52      *   sets this.tgt = tgt
53      */
54     public void setTgt(Node tgt) {
55         if (tgt != null) {
56             this.tgt = tgt;

```

```
57     }
58 }
59
60 /**
61 * @effects
62 *   return tgt;
63 */
64 public Node getTgt() {
65     return tgt;
66 }
67
68 /**
69 * @effects
70 *   if n.equals(src)
71 *   return true
72 * else
73 *   return false
74 */
75 public boolean hasSrc(Node n) {
76     if (n == null) return false;
77
78     return (src.equals(n));
79 }
80
81 /**
82 * @effects
83 *   if n.equals(tgt)
84 *   return true
85 * else
86 *   return false
87 */
88 public boolean hasTgt(Node n) {
89     if (n == null) return false;
90
91     return (tgt.equals(n));
92 }
93
94 @Override
95 public String toString() {
96     return toString(false);
97 }
98
99 public String toString(boolean full) {
```

```
100     if (full)
101         return "Edge("+src.getLabel()+" , "+tgt.getLabel()+" )";
102     else
103         return "("+src.getLabel()+" , "+tgt.getLabel()+" )";
104     }
105
106     @Override
107     public boolean equals(Object o) {
108         if (o == null || !(o instanceof Edge))
109             return false;
110
111         Edge e = (Edge) o;
112
113         return e.src.equals(src) && e.tgt.equals(tgt);
114     }
115 }
```

Bibliography

- [1] Aho, A. V. and Ullman, J. D. (1994). *Foundations of Computer Science: C Edition*. W. H. Freeman, New York, 2nd edition.
- [2] Alexander, C., Alexander, P. i. t. D. o. A. C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Shlomo, A. (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.
- [3] Boer, F. D., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C. C., Johnsen, E. B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., and Yang, A. M. (2017). A Survey of Active Object Languages. *ACM Comput. Surv.*, 50(5):76:1–76:39.
- [4] Booch, G. (1986). Object-Oriented Development. *IEEE Transactions on Software Engineering*, SE-12(2):211–221.
- [5] Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Conallen, J., and Houston, K. A. (2007). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional, Upper Saddle River, NJ, 3rd edition.
- [6] David J. Eck (2020). *Introduction to Programming Using Java*. Hobart and William Smith Colleges, 8th edition.
- [7] Dijkstra, E. (1979a). Programming considered as a human activity. In *Classics in software engineering*, pages 1–9. Yourdon Press, USA.
- [8] Dijkstra, E. (1979b). Structured programming. In *Classics in software engineering*, pages 41–48. Yourdon Press, USA.
- [9] Eisenbach, S., Khoshnevisan, H., and Vickers, S. (1994). *Reasoned Programming*. Prentice Hall Trade, New York.
- [10] Gamma, E., Helm, R., Johnson, R., Vlissides, J., and Booch, G. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Reading, Mass, 1st edition.
- [11] Gosling, J., Joy, B., Jr, G. L. S., Bracha, G., and Buckley, A. (2014). *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, Upper Saddle River, NJ, 1st edition.
- [12] Guttag, J. (1980). Notes on Type Abstraction (Version 2). *IEEE Transactions on Software Engineering*, SE-6(1):13–23.
- [13] Guttag, J. V. (2002). Algebraic Specification of Abstract Data Types. *Broy, M., Definert. E.,(eds.): Software Pioneers*, Springer.
- [14] Hoffer, J. A., George, J., and Valacich, J. A. (2013). *Modern Systems Analysis and Design*. Prentice Hall, Boston, 7th edition.
- [15] Kleppe, A. (2008). *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, Upper Saddle River, NJ, 1st edition.
- [16] Knuth, D. E. (1984). Literate programming. *Comput. J.*, 27(2):97–111.
- [17] Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 3rd edition.
- [18] Le, D. M., Dang, D.-H., and Nguyen, V.-H. (2018). On Domain Driven Design Using Annotation-Based Domain Specific Language. *Journal of Computer Languages, Systems & Structures*, 54:199–235.
- [19] Lilis, Y. and Savidis, A. (2019). A Survey of Metaprogramming Languages. *ACM Comput. Surv.*,

- 52(6):113:1–113:39.
- [20] Liskov, B. and Guttag, J. (2000). *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Pearson Education.
- [21] Nassi, I. and Shneiderman, B. (1973). Flowchart techniques for structured programming. *SIGPLAN Not.*, 8(8):12–26.
- [22] OMG (2015). Unified Modeling Language version 2.5. Technical Report formal/2015-03-01, OMG.
- [23] Oracle (2016). Java Platform Standard Edition 8 API Specification.
- [24] Oracle (2018). Java Persistence API Specification.
- [25] Ray, P. P. (2017). A Survey on Visual Programming Languages in Internet of Things. *Scientific Programming*, 2017:1231430. Publisher: Hindawi.
- [26] Red Hat (2017a). Bean Validation 2.0 (JSR 380).
- [27] Red Hat (2017b). Hibernate Validator.
- [28] Scott, M. L. (2009). *Programming Language Pragmatics*. Morgan Kaufmann, Amsterdam ; Boston, 3rd edition.
- [29] Sebesta, R. (2015). *Concepts of Programming Languages*. Pearson, 11th edition.
- [30] Sommerville, I. (2011). *Software Engineering*. Pearson, 9th edition.
- [31] Tomassetti, F., Bruggen, D. v., and Smith, N. (2016). *JavaParser: Visited*. Leanpub.
- [32] Trois, C., Del Fabro, M. D., de Bona, L. C. E., and Martinello, M. (2016). A Survey on SDN Programming Languages: Toward a Taxonomy. *IEEE Communications Surveys Tutorials*, 18(4):2687–2712. Conference Name: IEEE Communications Surveys Tutorials.
- [33] Wallace, D. and Fujii, R. (1989). Software verification and validation: an overview. *IEEE Software*, 6(3):10–17. Conference Name: IEEE Software.