Emperical Semantics for Object-Oriented Programs

Jan Bergstra *Marijke Loots*

Preprint nr. 007 July 1999

Artificial Intelligence Preprint Series

Empirical Semantics for Object-Oriented Programs

J.A. Bergstra[1] & M.E. Loots[2]

June 28, 1999

[1]University of Amsterdam, Programming Research Group, & Utrecht University, Department of Philosophy, Applied Logic Group, email: Jan.Bergstra@phil.uu.nl.
[2]Tilburg University, Faculty of Philosophy, email: M.E.Loots@kub.nl.

Contents

Part I

FHN and other preliminaries

Chapter 1

Introduction

1.1 Single-threaded, single-inheriting

JavaCck, (Java Class Construction Kernel) is introduced as a subset of Java™ (Java is a trademark of SUN Microsystems). It serves as a vehicle for the description of object-oriented programming notations.

The focus is on object orientation in a sequential or single-threaded setting and with single inheritance. The second restriction points in the direction of Java[1], the first restriction calls for a further subsetting of Java. JavaCck is a very restricted Java subset concentrated on the use of object-oriented constructions.

The object-oriented features of JavaCck will be studied in detail. JavaCck is considered a very workable setting to discuss the principles of object orientation. Discussing object-oriented programming from the perspective of a single language has advantages and disadvantages, the evident disadvantage being the bias towards a particular language design. The advantage, outweighing the disadvantages, lies in the possibility of dealing with the details of object orientation and of moving beyond the popular generalities. This book is meant for readers with a professional interest in object-oriented programming, in particular computer science students[2] and researchers[3] in programming language design and implementation. The novelty of the book lies in the following three aspects:

1. The folder hierarchy notation FHN is introduced. FHN provides a most explicit technique to describe class families.[4] FHN is introduced and its

[1] We refer to [1] as an original source for Java, and to [7] and [10] as representatives for the fast growing volume of text books on Java.

[2] The book is intended to be useful for the teaching of object-oriented programming to students having no preparatory programming experience in Java or in other programming languages. Since it is not a self-contained course, additional information about the various language features will be needed. It is assumed that teachers using this material consider it attractive to produce such material themselves.

[3] For research the folder hierarchy notation and the empirical semantics methodology provide ingredients that may prove to be useful in many other circumstances.

[4] Rather than the word 'program' the phrase 'class family' will be used.

meaning is explained by means of an informal explanation as well as by an algebraic specification.

2. JavaCck is identified as a kernel language for object-oriented programming, very suitable for semantic analysis.

3. Empirical semantics is introduced and used as an effective method to describe the semantics of an object-oriented program notation.

For the textual representation of JavaCck a so-called class organization syntax is proposed. The particular class organization syntax used here is FHN. The text contains an extensive collection of input/output relations for JavaCck class families that should in principle be sufficient to determine the greater part of JavaCck semantics.

Empirical semantics aims at giving a description of the meaning of a program notation or a subset of a program notation by providing a systematic collection of examples of behavior. Emprical semantics can be contrasted with more formal methods to provide the meaning of a program notation, such as projection semantics. The authors are unaware of any existing systematic scientific methodology for developing and presenting an emprirical semantics for a program notation. Therefore the title refers to an empirical semantics project, and does not claim to cover the concept of 'empirical semantics'.

It is more than likely that a more systematic approach will yield a far more informative body of knowledge than the collection presented below.

1.1.1 Acknowledgements

The authors acknowledge discussions with and/or input/support from: Kees van den Berg, Eggie van Buiten, Andre van Delft, Lou Feys, Jan-Friso Groote, Dimitri Hendriks, Wim Hesselink, Bart Jacobs, Hayco de Jong, Paul Klint, Andre van Kooy, Karst Koymans, Kees Middelburg, Pieter Olivier, Alban Ponse, Piet Rodenburg, Vincent van Oostrom, Piet Rodenburg, Jerzy Tiuryn, Jaroslav Usenko, Freek Wiedijk.

1.2 Project Rationale for Empirical Semantics

Any empirical semantics project for a language or a part of a language will need some general rationale. The main characteristics determining and limiting the rationale of empirical semantics are listed below.

Irrational programming. All semantic aspects of a language are equally important, and do not depend on the rationality or style of the programs used or needed to illustrate such aspects.

Unbiased observation. The single most important source of information *not* available to the empirical semantic analyst is his or her opinion on 'what the meaning of the program notation ought to be'. There are only facts,

there is no apriori rationale for any system behavior which allows the analyst to ignore his/her observations, however implausible these observations may appear.

Random experimentation. The design of experiments is only guided by the intention to maximally clarify the meaning of all possible program notations. No particular language design bias allows one to ignore aspects that might be considered of lesser importance on the basis of an assessment of the relative merits of different aspects of a language.

Simple experiments. The goal to keep the experiments simple is achieved if processing the examples is well within the reach of a human being with an adequate understanding of the language. This may not always be a feasible constraint, but it has been a restriction imposed on the project below.

Pre-formal semantics. Empirical semantics is best developed before a formal semantics is designed. Different formal semantic models of a program notation will have to comply with the same body of experimental facts (i.e. documented empirical semantics projects.)[5]

Readable project report. The project report of an empirical semantics project must be informative to new users of its target notation. Experimentation is not needed to either understand or appreciate the project report.

1.3 Status of the empirical data

All experiments reported below have one of the following forms: (i) Program P is not accepted by the compiler (i.e. the version of JDK 1.2.1 running on the authors' equipment during April 1999 at the Department of Philosophy of Utrecht University.), (ii) Program P is not accepted and the error message E is produced, (iii) Program P is accepted and its execution diverges, (iv) Program P is executed and its execution produces the following outputs (written on the screen), including error reports. The data are empirical and most programs were run several times.

This collection of data can be used as a first confrontation with object-oriented concepts (such as represented in Java). No preliminary programming knowledge is presupposed for someone to appreciate these data. In this raw form they constitute a basis on which a more imaginative and complete formal or informal model of the meaning of programs can be developed when needed. The 'observations' reported below can have different interpretations. First of

[5] A projection semantics can be viewed as a concise and scientific explanation of the body of data emerging from a projection semantics project. The need for such an explanation arises when either the body of data is too large to be useful or the requirements of program correctness, behavior prediction and behavior analysis are so sharp that more systematic knowledge of the meaning of programs is vital. For the arguments in favor of projection semantics we refer to [3].

all they document observed system behavior for a very particular configuration, called PSC (production site configuration). PSC involves all parameters having to do with the versions of software, modes of installation, type of machinery, choice of operating system etc. at the time of producing this document. PSC is more or less unique. It is unlikely to have an exact copy anywhere else in the world.

1.3.1 Verdicts for empirical data

Let $O(X)$ be an observation on JCF (program) X. Here are a number of possible ways $O(X)$ can be viewed, the views being expressed as verdicts or interpretations of particular observations:

manifestly correct behavior. The observation documents a property of Java execution which should be considered valid from all meaningful perpectives.

manifestly version-dependent behavior. Behavior may be specific for a certain implementation, not compliant with a 'mainstream interpretation' of the program notation.

unspecified behavior. Unspecified behavior occurs if output is generated in circumstances where the specification makes no definite predictions on how execution must take place. To determine whether behavior is unspecified reference must be made to some language description, reference manual, language standard or other document with sufficient authority and precision. Computer programs should not depend on unspecified behavior (in principle). Therefore it is irrelevant to 'know' unspecified behavior in detail. It is very relevant to know when behavior should be considered specified. This is a difficult subject, however, and is not a first-class topic for presenting the principles of a language.

marginal behavior. The behavior is plausible and defendible (on the basis of existing documentation) but its importance for software engineering should be considered minimal. Some programs will not emerge if rigorous programming methods are applied; unpredicted execution of such programs may be considered a 'small evil'.

conceptually incorrect behavior. Execution may produce results demonstrating conceptual flaws in the language design. For this category to be potentially non-empty it is crucial to view the language as a realization of a body of concepts and mechanisms having an independent conceptual status. Spotting a conceptual error may (but need not) lead to a disagreement with language designers and compiler writers on how to realize the afore-mentioned body of knowledge 'in software'.

manifestly incorrect behavior. The obervation indicates that 'something is wrong'. This may range from a (known or unknown) compiler bug to a local system error.

1.3.2 Synthesis of an initial mental picture

The very ability to distinguish between the above interpretations (in a specific case) is a highly non-trivial competence. The position taken here regarding these matters is that the data presented below constitute a coherent body of information. This body of information can be processed into a coherent mental picture by means of human inductive reasoning capabilities. On the basis of such a mental picture further refinements are possible, leading to a classification of behavior in the classification mentioned above. In the construction of the mental picture all data are by default classified as *manifestly correct*.

Fuzzy borders

The mental picture should not be misunderstood as a fully reliable ability to predict program behavior in all cases similar to the ones listed below. Different readers will achieve different degrees of this ability. The stable situation for a Java programmer is a limited ability to predict program behavior and some awareness of these limits. The mental picture involves a fuzzy demarcation between predictable and unpredictable, the borders being different from person to person.

Possibilities are necessities

Because single-threaded, single-inheriting Java is a deterministic language, the assumption that all observations reflect reasonable system behavior implies the far more significant assumption that all observations reprepresent necessary system behavior. The implication follows from the complete determination of program execution in case a program is deterministic and definitely within the part of the syntax for which reasonable behavior is defined.

1.3.3 Refinement of the initial mental picture

After an initial categorization of behavior has been made, it may turn out that some observations will have to be classified in another category. An initial mental picture of sequential object orientation using single inheritance can be (but need not be) synthesised from the 'use cases' listed in the empirical semantics document. Subsequent refinement (potentially involving revision) of the picture requires an investigation of many other sources of information. Producing a mathematical semantics for some fragments of Java, however, can be done solely on the basis of a mental picture generated by the empirical semantics report.

1.3.4 Specific for OO?

The question may be raised whether empirical semantics is specifically suited for object-oriented programming. The authors believe this to be the case. The conceptual power and 'market value' of object-oriented programming is connected

with strong mechanical intuitions regarding taxonomic structures of active entities.

Object-orientation is primarily conceptual

Object-orientation is based on very general principles of hierarchical taxonomic structures. Taxonomic structures are classic in many disciplines, most prominently in biology. Novel for object orientation in programming is the connection with dynamic aspects of objects (classified elements). The combination of structural taxonomy and dynamic aspects in programming rests on a number of computational intuitions. These intuitions together with an understanding of the interaction between structural taxonomy and behavior have matured over many years, the process of maturation surfacing in successive programming notation designs.[6] The mental picture mentioned above provides an explicit (though informal) representation of these intuitions, framed in the format of a subset of Java. The very appeal of that mental picture is the selling-point of object orientation. Introducing object-oriented programming along the lines of empirical semantics should reinforce this mental picture in the strongest possible form.

Other programming techniques such as multi-threading can be viewed as tools to bring certain functionalities to expression, the functionalities being easy to grasp or to put into words. Although the functional options for object-oriented programming are enormous, short object-oriented programs cannot be used to demonstrate the practical value of (or even need for) object-oriented programming convincingly. All that can be done by means of short programs is to clarify how things work, not why some things are practically relevant in a very clear-cut way.

Single-threaded OO-systems are deterministic

The restriction to single-threaded systems makes program execution deterministic. As a consequence the initial mental picture will be informative about what machine execution of a JCF should yield, not only about what it might produce.[7] This argument allows one to study the empirical semantics of OO-programming in the single-threaded case without much regard for language documentation (at least initially). In addition, this argument provides a strong motivation for restricting JavaCck to the single-threaded case.

1.4 Technical details

This text contains many example programs and outputs from running the programs. All examples are supposed to be fit for manual computation, enabling

[6] It is not implied that Java constitutes an optimal combination of OO-concepts in all respects. This is also a matter of taste.

[7] In the case of multi-threaded systems, empirical semantics should be applied with far more care because observations may only demonstrate possible (acceptable) system behavior and be uninformative about the necessities of system behavior.

one to verify a given semantic story against the observations it should explain.[8] As regards the project of empirical program language semantics the following remarks can be made:

- All computations have been performed on the same 5-year old configuration. Modern machines may be expected to be some 10 times faster. The overall picture will not change. All data ought to be correct for JDK 1.2.1 in particular the correct outputs that are not influenced by the machine limitations.) Substantial differences between JDK 1.1.6 and JDK 1.2.1 observed while processing the examples.

- The description of JavaCck constitutes no criticism of Java. In fact, the JDK 1.2.1 of the JavaCck part of Java strikes the authors as being remarkably logical and systematic. It should be stressed that JavaCck is in no way meant to be a program notation for any other purpose than the empirical semantics project.[9]

- The collected data may be compiler dependent and/or platform dependent. It requires special care to decide whether the observations are 'reasonable' or whether compiler and/or platform defects (or expected limitations) are being observed.

- A program is said to be accepted by the compiler if it is processed without generating any complaints or warnings from a situation in which no class files have previously been generated (for the material classes.) Although interesting in its own right, the learning behavior of the compiler is not an object of study.

1.5 JavaCck and Class Writing

JavaCck is a small subset of SUN Microsystems' extremely popular program notation Java. JavaCck is introduced here with two objectives in mind: (i) reduction of the complexity of Java for teaching purposes, (ii) greater comprehension of at least part of the language. JavaCck is as much a subset of C^{++} (albeit with some syntactic modifications) as it is of Java. Java's advantages over C^{++} are mainly related to aspects not contained in JavaCck.[10]

[8] Of course the human processor intending to compute a case by hand needs to use a theory. That theory is also needed to understand the complexity of the case at hand. The statement that these examples are amenable to human processing cannot be made explicit in any informal setting. Still, the statement seems to be intuitively meaningful.

[9] The information stated in this paper has been retrieved by means of experimentation with SUN JDK 1.1.6, using a SUN 10 server and a Tektronix X-terminal. Most examples have been run with JDK 1.2.1 well.

[10] A key aspect of full Java is the presence of a class library with external classes that need not be in Java itself. By extending such classes, and typically by overrriding some of its methods, native code can be used from Java with a minimal pollution of the Java code written by the occurrence of 'other program notations'. Extending a class written by someone else is a clear and convincing case of software re-usage.

JavaCck being a C++ dialect, the references that have been most influential for JavaCck are [9] (for C) and [6] for C++. Of course the influence of both SIMULA 67 and Smalltalk on object-oriented programming are undisputed. C++, however, marks the stage of seemingly unconstrained industrial acceptance of object-oriented programming.[11] Therefore it comes as no surprise that Java has mainly been influenced by C++. Since JavaCck focusses on the object-orientation within Java, it is very close to a (small) subset of C++ in addition to being a subset of Java.

The empirical semantics approach makes use of new ideas and terminology necessitating additional explanation. We list:

- As texts in JavaCck are collections of classes, the term 'class writing' is used for the production of such texts.[12]

  Class writing is a neutral term, devoid of any a priory implication of objectives, purposes, goals, quality criteria, design techniques etc. The purpose of class writing may be to find out to what extent a compiler meets its stated specifications, to study error messages generated in a particular instance, to settle a question regarding the language syntax, or to obtain a rough feeling about its semantics by means of experimentation with the language environment at hand. This text is restricted to the objective last-mentioned.

- The principal product of JavaCck class writing is a JavaCck Class Family (JCF), which is a Java Class Family at the same time of course.

- JavaCck is provided with an explicit class organization syntax, prescribing how classes can be organized in a hierarchical file system.

1.6 JavaCck Class Construction Syntax

The class construction syntax of JavaCck is obtained from the class construction syntax (usually simply called the context-free syntax) of Java by deleting many features of Java.[13] No attempt is made to equip JavaCck for programming purposes. JavaCck can be structured as an increasing family of subsets of Java, all subsets obeying a number of restrictions. The restrictions imposed on JavaCck languages are these:

1. All method calls to methods outside JavaCck programs are made by members of classes of a package `tjvap`. All of these members are void and static methods, being members of final classes. Being void, these methods will never return any value to class control, the consequence being that the

[11] It is astonishing that the complexity of C++ has not been a far greater impediment to its industrial acceptance that it actually is (reported to have been).

[12] No implication exists that classes are programs or programs are classes. The relation between class writing and programming is left open, albeit that class writing is best regarded as an aspect of programming rather than conversely.

[13] As a source for information on Java the authors have mainly relied on: [10].

external behavior of a JCF is a computable sequence of output actions (method calls) to `tjvap` classes. This sequence can terminate properly, terminate in divergence (D), terminate in error (M) or proceed infinitely. Semantically, JavaCck class families denote programs with void actions only, using the terminology of program algebra. From the point of view of program algebra TCF's are (translatable into) programs, never taking any external input and never relying on an the effect of an external test.

2. Booleans are used extensively. Except for integers, no other data types are used. Integers can only be initialized by means of a constant (no assignments or operator)

3. Command line input is not used.

4. Inner classes are omitted.

5. The use of access specifiers is minimized; the specifier 'public' is only used if necessary.

6. JavaCck makes no use of multi-threading (the class Thread is never extended, the interface 'Runnable' is not implemented.).

7. JavaCck ignores exceptions, and disallows explicit calls for garbage collection.

8. No external classes can be extended in JavaCck languages, thus making the use of the JDK class library impossible.

9. JCF's must contain a class called `c` with a method `m()`. In addition there should be a class `s` containing a single method `main()`, having as its only effect a single call of `c.m()`. Execution of a JCF is started by means of a call to that program.

Chapter 2

An incremental feature collection

The program notation JavaCck is a small (and fuzzy) subset of Java, demarcated by means of informal objectives rather than by means of a fully explicit sublanguage syntax. JavaCck can be segmented into various levels, the levels being described in terms of objectives and qualitative characteristics. The combination of objectives and qualities does not always lead to a watertight classification of particular JCF's. However, the aim of this chapter is to identify the features rather than to outline a conclusive classification of sublanguages.

2.1 Syntax, concepts and features

Syntax determines the collection of texts that qualify as correct JCF's. Correctness is defined as being accepted by the compiler, no static errors being reported. Run-time errors relate to dynamic semantics rather than to syntax. Syntax determines which language elements can be combined into meaningful units. Syntax is usually described by means of a grammar and rules for generating correct expresssions from that grammar. We will not provide a grammar for JavaCck classes and class families. All examples below are grammatically (or syntactically) correct (unless stated otherwise, mostly with an indication of the error message returned by the compiler).

As a program notation, JavaCck will allow a programmer to express systems (algorithms) by means of a range of computational concepts. JavaCck is a sequential (single-threaded), object-oriented (and single-inheritance based) language. Characteristic of such a language is the combined presence of at least the following computational concepts.

explicit flow of control. In particular: iteration, recursion, conditional branching, externallly visible actions.

explicit object/value manipulation. In particular: references, object creation, types, assignment, method application, object casting

object typing. In particular: built-in value types, built-in classes, defined classes and interfaces.

class definition. In particular: ground class definition (no explicit super classes) and class inheritance.

explicit scope control. In particular block structure (and local variables), access specifiers, static versus non-static.

2.2 Features and concepts

A feature is a coherent collection of syntactic ingredients or forms, used to express certain computational concepts. To illustrate the notion of a feature some examples are mentioned below. A typical feature is the existence of class extensions (CE). This feature is syntactically visible in a keyword that expresses one class extending another one, (`class A extends B {..}`). Class extension is one of the features for class definition. Abstract class definition is another feature for class definition (`abstract class A {..}`), interface implementation is yet another feature for class definition (`class A implements I {...}`). Class extension is a form of class definition using inheritance.

Another feature is the existence of class instances (CI), (`A x = new A();`). Class instances are the so-called objects to which object-oriented programming refers. Class instantiation is a feature for object creation. The while-construct (`while(p){..}`) is a feature for flow control.

2.2.1 Open-ended potential

The concepts being so general, the list of (options for) features realizing such concepts is open-ended. For that reason it is implausible, and even impossible to provide an exhaustive discussion of all features for a certain concept. For each concept quite an number of features are provided in Java, thus enabling a user to express programs in a flexible way.

Why features

Features are useful for defining subsets of a language. Any collection of features can be taken to specify a subset of a program notation. Natural subsets of a program notations will usually correpond with a collection of features.

In its turn, the importance of subsets is simply the fact that sophisticated information can more easily be obtained for subsets of a language than for the entire language. For languages of the complexity of Java, logical analysis should proceed incrementally in stages, taking ever larger feature collections as a point of departure.

Feature combinations

Sublanguages correspond to combinations of features. The features will be briefly discussed here, further comments regarding the features being given in connection with the portfolio of examples below.[1] Feature combinations may be far more complex than individual features seem to be. Combining CE and CI for instance requires clarification concerning inheritance at the level of objects.

Feature interaction

Features often serve a clear purpose, even in the absence of other features. In the presence of other features many forms of interaction emerge for which an implementation should solve a semantic problem. Unless particular combinations are disallowed by the compiler, the empirical semantics project should investigate their properties.

2.3 A survey of features

The features below are not strictly clustered along the lines of the division of concepts as given above. The merit of a meticulous distinction in features is that sophisticated definitions of language subsets can be generated on the basis of relevant feature combinations.

2.3.1 A 'minimal feature set': BJC

BJC is Basic JavaCck. It is a very minimal subset just sufficient to write the simplest class families. The feature VCM (void class methods) is implicit in BJC. It allows one to write a number of classes with void static methods only. Methods can be called inside a method body, the only control combinator being ';'.[2] As a consequence BJC admits recursion but not iteration. BJC serves as a basis, its extensions being found by admission of additional features from the listing below.

Class methods are connected with classes. Class method definitions are part of class definitions. Class method definitions always carry the specifier `static`. Methods can have parameters; parameter lists are explicitly given as a list of pairs of types and names. In case a method has zero parameters an empty parameter list is still mandatory. Types are either 'boolean' or they refer to some class that is introduced in the JCF. In BJC only boolean parameters for methods are allowed. If a method is used inside its own class, the mere mentioning of the method name is sufficient; if used outside its own class, the

[1] The meaning of the various features as well as their syntax will become clear only later. It seems reasonable to say that none of the features mentioned is an artefact of the Java language design. In all cases the features can be given a general meaning independent of the details of Java.

[2] In most program notations ';' is referred to as sequential composition. We prefer to call it concatenation or text-sequential composition, because of there being no definite relation with the progress of time.

method is called by prefixing its name with the originating class name. Class name and method will be separated by the selector dot ('.').

2.3.2 Extensions of BJC

All features are placed on top of BJC. Some features, however, are meaningful only on top of earlier ones.

Sequential control: WHI, CND, bRCM, bASG and bRET

SCS[3] represents the collection of sequential control structures of Java. Of these JavaCck allows just two: WHI and CND. WHI represents the while construct, CND the conditional constructs (if then, if then else), bRCM denotes the option of using and defining boolean (returning) class methods; bASG allows the use of assignment to boolean variables. bRET denotes the boolean-returning construct. bRET is needed to provide a last instruction in a boolean-returning or object-returning method; it takes the form `return bExp;` with `bExp` a boolean expression. bRET is only useful in the presence of bRCM or bRIM as mentioned below.

Boolean class variables and instances variables: bLV, bSV, bIV

Three types of boolean variables can be used: local variables, used and declared within a method body; static variables for a class, and instance variables connected with class instances. The 'b' indicates JavaCck's restriction to boolean values.

Whether a name in a class interface listing refers to a variable or a method is always simple to decide: variables have no parameters, methods do.

BJC+bLV denotes a language fragment of JavaCck allowing local boolean variables on top of the facilities of BJC, BJC+bSV allowing the use of static boolean variables. Variable declarations (static or class) are part of class definitions. Boolean variables are introduced (declared) as boolean typed identifiers (`boolean v;`). Simultaneously with their introduction, local variables can be initialized (`boolean v = Exp;`). Local variables exist within a local scope inside the body of a method. Access to local variables from within the scope of their definition is obtained by means of the use of their name.[4]

Boolean class variables are connected with a class, the static variables appearing in the class definition (`static boolean v;`). From inside the class, access to a static class variable can be obtained by using its name; from outside the class, access requires prefixing the name with the class name (of the class to which the static variable belongs) separated by a selector dot (`className.v`). Boolean class variables can be initialized (`static boolean v = Exp;`). The

[3] SCS denotes all sequential control structures of Java. In addition to WHI and CND, SCS (for full Java) includes switch statements, exceptions, for statements and jump statements.

[4] When inside the body of a method `m1()` a call of method `m2()` is executed, the local variables of `m1()` are temporarily out of scope until the call of `m2()` has returned. This constitutes a major difference with recursion in the style of ALGOL60.

initialization actually takes place when execution first encouters a need to use the class. The boolean class variables are initialized in the order of definition.

Boolean instance variables (also called boolean instance fields) are connected with all instances of a class. The introduction of a boolean instance variable is part of the class description (`boolean v;`). The distinction between class variables and instance variables is made by the presence or absence of the keyword `static` for the variable, the static ones being class variables, the others being instance variables. The difference with local variables is that those are defined within a method body. Instance variables can be initialized. Initialized instance variables get a value at the time of the construction of an instance. Access to instance variables is similar to access to class variables from within 'their' class. From within other classes, access is granted by mentioning the name of the object (instance) from which the variable must be taken. The instance variable serves as a selector applicable to instances. Instance and selector are separated by the selector dot (as in the case of static variables).

Object variables: oLV, oSV, oIV and oASG

Variables for objects can be local (oLV), static (oSV), instance (oIV). Assignments to variables of arbitrary object types are permitted by the feature oASG. The rules of the game for object variables are similar to those for boolean variables. Of course, an object variable definition requires an object type. Classes serve as object types in Java.

Object variables can obtain values only through the construction of objects, either default constructors or explicit constructors. Objects can be transformed by method applications. The syntax for object variables of the three kinds mentioned is similar to that of boolean variables, the type boolean being replaced by the class name.

Class instantiation: CI, VIM, bRIM, oRET oRIM, oRCM, CON, THI and Sup

CI indicates the option of instantiating a class, thus obtaining an object (`className instName = new className();`). VIM represents the possibility of introducing (and using) void instance methods. An instance method is a method to be executed 'by an instance'. That instance is the target of the method. All instances of a class can execute the same instance methods and provide access to the same instance fields. Instance methods and class methods can be distinguished according to the (absence) of the keyword `static`.

bRIM allows the declaration and use of boolean returning instance methods, oRIM indicates the declaration and use of object-returning instance methods. Non-void methods must terminate the execution of their method body by a return instruction returning an expression (object/value) of the right type. oRCM represents the option of using and declaring object-returning class methods. CON represents the use of constructors (explicit constructors that replace and/or override the default constructor), THI allows the use of the keyword

`this`, referring to the target of a method from its body. Sup refers to the use of the keyword `super` as a notation for the object from which another object immediately inherits. Like boolean variables, object variables are declared as typed identifiers, initialization at declaration being an option.

Instance methods are applied to objects of the class in which they have been introduced.[5]

Class extension: CE, SUP, CAS, EQU and INOF

CE allows the use of class extensions. Extended classes inherit class fields (static variables) from their super class. Instances from extended classes are extensions of instances of the super class. SUP introduces the calling of superclass constructors. EQU allows the use of the built-in equals operator. INOF allows the use of the 'instance of' test. This test allows a run-time query on the class membership of an object. CAS denotes the use of the casting operator. Casting moves an object of a superclass to a subclass.[6]

Other control structures: ASP and IF

Access specifiers are `public, private, protected, default` for class variables (both boolean type and object type) and instance variables of both types. In addition the same access specifiers can be used for class methods and instance methods of both types. For classes, class variables and instances variables, the access specifier `final` can be used. ASP indicates the use of all access specifiers. IF allows the use of interfaces.

2.3.3 Miscellaneous: OVL, OVR, SCOj and GC

Some mechanisms of JavaCck are implicit in the sense that no specific syntax is devoted to them. Typical examples are: (i) overloading (OVL) which allows the presence of operators differing only in the types (classes) of their arguments, (ii) overriding (OVR) allowing extending classes to redefine static fields and methods, as well as instance fields and methods,[7] (iii) SCO refers to scoping. SCOj stands for the particular scoping rules used for JavaCck, (iv) GC stands for garbage collection, a feature responsible for Java's ability to reclaim memory space which cannot be accessed anymore.

[5] In the famous SMALLTALK terminology the instance is sent a message (representing the method to be applied).

[6] Casting affects the interpretation of instance method application and instance variable selection. The definition of instance variables and instance methods may vary from class to class. An object can only be cast to a class of which it happens to be an instance. The preliminary use of the INOF feature protects the class writer against dynamic errors that might occur in this connection. Casting is needed if an object is named by a variable for a subclass of its 'home' class.

[7] Special attention will be paid to the interaction between CAS and OVL and between CAS and OVR, the results being somewhat unexpected.

2.4 The hierarchy of features

The mutual dependency of features leads to a hierarchy. Features are stacked on top of one another. The dependency graph is generated by pairs of features (A_1,..,A_n) < B with A_1,..,A_n being prerequisites for B. A (non-exhaustive) listing of such pairs is helpful:

CE < OVR, (CI,CE) < INOF, CI < IN, CI < EQU, (CI,CE) < CAS, CI < CON, CI < Sup, (CI,CON) < SUP, CI < 'all features involving "i" and "o"',

2.4.1 Sublanguages of JavaCck

On the basis of the inventory of features just defined many sublanguages of JavaCck can be identified, the smallest being BJC. Some examples are mentioned to illustrate the notation. The task of providing a logical semantics of JavaCck, as distinct from providing an empirical semantics, can be modularized along the lines of this schematization of sublanguages. A formalized semantics can be determined for increasingly complex notations of the form BJC+X+Y+Z+$\cdots$.[8]

- BJC+bLV+bCV+bCM+SCS allows the use of boolean variables, known from procedural languages. Object-orientation plays no role in this notation.
- BJC+bCV+CE is the minimal class notation in which inheritance can be used.
- BJC+CI is the minimal notation that allows the use of objects.
- BJC+CI+bIV suffices to illustrate a significant phenomenology concerning classes and objects.
- BJC+CI+CE+CON+RET+bLV+bCV+bIV+oLV+oSV+oIV+bRCM +bRIM+oRCM+bRCM denotes a class notation in which most of the object-oriented features of Java can be adequately illustrated.

The rationale for introducing the hierarchy of sublanguages is that each of them can (in principle) be provided with a definite syntax (after distinctive decisions have been made) as well as of a definite semantic description. The semantic descriptions can be given in many styles and in many forms. It is practical to regard semantic descriptions of full Java as a family of semantic descriptions of a hierarchy of Java sublanguages. By considering Java fragments that pose only limited semantic complexities it is also possible to compare different semantic styles and formats and to obtain a thorough understanding of the various methods.

[8] Projection semantics is a possible technique for doing so.

Chapter 3

Folder Hierarchy Notation

Being a subset of Java, JavaCck classes are produced as text files having a `.java` name extension. These files are collected in a package. Named packages of files with java classes can be part of a JCF. The folder hierarchy notation(FHN) allows one to denote particular JCF's in full information outfit. The term folder is used rather than package because some folders denoted in FHN may fail to qualify as Java packages, for instance if the included files in no way contain Java classes. FHN allows the notation of named and structured JCF descriptions, thus allowing (by juxtaposition) the description of a portfolio of JCF's. FHN provides a syntax for the description of folders. Folders can be empty, or they can contain a number of files carrying names, files being organized sequentially, folders hierarchically. In the description of FHN A, A_i ranges over texts and B, B_i ranges over folders. The size of a folder is its number of elements at top level. In the case of a JCF texts A, A_i are chosen to be flat file representations of classes. Concatenation of texts is denoted with '$*$'.

3.1 Folder Hierarchy Notation Primitives

The primitives of FHN are these:

- $\emptyset$ represents the empty folder. Its size is 0.
- `file:` *fileName*(A), denotes a file named *fileName* containing the character sequence A. The file `file:` *fileName*(A) is itself a folder of size 1.
- `folder:` *folderName*(B), denotes a folder named *folderName* containing the folder B. The folder `folder:` *folderName*(B) has size 1.
- Given folders collection B_1 and B_2, $B_1 \cup B_2$ is the union of the two folders. There are some constraints:

- In this union the folders of B_1 that have names also occurring as B_2 names are united. For a particular name a folder contains at most one folder with that name.
- The situation for files is similar but not identical. If B_1 contains a file `file:`$fileName(A_1)$ and B_2 contains a file `file:` $fileName(A_2)$, the combination $B_1 \cup B_2$ will contain a file `file:` $fileName(A_1 * A_2)$,($*$ denoting string concatenation.) This rule prevents folder union from being commutative. Again a folder can contain (at each level) at most one file under some given name.
- If B_1 contains a folder `folder:`$Name(B)$ and B_2 contains a file `file:` $Name(A)$, the combination $B_1 \cup B_2$ will contain the folder but not the file. This rule guarantees that at any level a folder contains at most one item (folder or file) under a given name.

An FHN-JavaCck expression is an FHN-expression with all texts A listed in files being class descriptions. For such an expression to be syntactically correct the empirical criterion is that it is accepted by the compiler of JDK 1.2.1. FHN-expressions for JavaCck will be used to represent JCF's.

A path is a sequence of names written as $name_1/name_2/../name_k$. Paths can be used to indicate files of folders within an FHN-expression or a JCF. With B a JCF and p a path B.`file`(p) denotes the text (if any) within B with path p leading to it. Similarly B.`folder`(p) denotes the folder (if any) reached along path p. Thus `file:` $fileName(A)$.`file`($fileName$) $= A$ etc. If no file name or folder can be found using the selection operator the meaningless value M is produced.

In addition to the constructors and selectors mentioned above FHN admits a subtraction operator $\Gamma - \{p_1, \ldots, p_k\}$ removing all files and folders named in the set $\{p_1, \ldots, p_k\}$ from the JCF Γ.

3.2 Equations for FHN

The rules for the folder hierarchy notation FHN can easily be summarized in a set of conditional equations[1]. The use of the equations is as follows: in the process of writing a JCF, named files and named folders can be introduced in successive stages. The equations describe how to combine parts of files and folders into complete files and folders. This process has to be performed before deletion of named folders or files is attempted. The flexibility of FHN is helpful when a portfolio of JCF's is to be denoted.

In these equations x, y range over names of files/folders, A, A_i range over texts (file contents), B, B_i range over folders, p, p_i range over paths.

[1] FHN is called a notation in spite of its being an algebra according to even the most restricted definitions. Although FHN is an algebra, its equations are neither deep nor difficult. The transformation rules expressed by the equations are all fairly obvious. FHN is a bookkeeping device and no more than that. This lack of intrinsic interest of the FHN equations motivates the decision to call it a notation rather than an algebra or a calculus.

3.2.1 Equations for folders

$$\emptyset \cup B = B, B \cup \emptyset = B$$

$$(B_1 \cup B_2) \cup B_3 = B_1 \cup (B_2 \cup B_3)$$

$$\texttt{file} : x(A_1) \cup \texttt{file} : x(A_2) = \texttt{file} : x(A_1 * A_2)$$

$$\texttt{folder} : x(B_1) \cup \texttt{folder} : x(B_2) = \texttt{folder} : x(B_1 \cup B_2)$$

$$\texttt{folder} : x(B) \cup \texttt{file} : x(A) = \texttt{folder} : x(B)$$

$$x \neq y \rightarrow \texttt{folder} : x(B_1) \cup \texttt{folder} : y(B_2) = \texttt{folder} : y(B_2) \cup \texttt{folder} : x(B_1)$$

$$x \neq y \rightarrow \texttt{file} : x(A) \cup \texttt{folder} : y(B) = \texttt{folder} : y(B) \cup \texttt{file} : x(A)$$

$$x \neq y \rightarrow \texttt{file} : x(A_1) \cup \texttt{file} : y(A_2) = \texttt{file} : y(A_2) \cup \texttt{file} : x(A_1)$$

3.2.2 Equations for the remove operator

The equations for the remove operator can be defined on top of this basis (p ranges over paths):

$$B - \{p_1, ..., p_{n+1}\} = (B - \{p_1\}) - \{p_2, ..., p_{n+1}\}$$

$$(B_1 \cup B_2) - \{p\} = (B_1 - \{p\}) \cup (B_2 - \{p\})$$

$$\emptyset - \{p\} = \emptyset$$

$$x \neq y \rightarrow \texttt{file} : x(A) - \{y\} = \texttt{file} : x(A)$$

$$x \neq y \rightarrow \texttt{folder} : x(B) - \{y\} = \texttt{folder} : x(B)$$

$$\texttt{file} : x(A) - \{x\} = \emptyset$$

$$\texttt{folder} : x(B) - \{x\} = \emptyset$$

$$\texttt{file} : x(A) - \{y/p\} = \texttt{file} : x(A)$$

$$x \neq y \rightarrow \texttt{folder} : x(B) - \{y/p\} = \texttt{folder} : x(B)$$

$$\texttt{folder} : x(B) - \{x/p\} = \texttt{folder} : x(B - \{p\})$$

3.2.3 Defining equations for the selectors

The equations for both selection operators can be given, using the remove operator. Four equations describe the 'positive cases'.

Positive cases

$$(\texttt{file} : x(A) \cup (B - \{x\})).\texttt{file}(x) = A$$

$$(\texttt{folder} : x(B_1) \cup (B_2 - \{x\})).\texttt{folder}(x) = B_1$$

$$(\texttt{folder} : x(B_1) \cup (B_2 - \{x\})).\texttt{file}(x/p) = B_1.\texttt{file}(p)$$

$$(\texttt{folder} : x(B_1) \cup (B_2 - \{x\})).\texttt{folder}(x/p) = B_1.\texttt{folder}(p)$$

Negative cases

In all other cases an error is to be produced. This requires a disapointingly large numbers of defining equations:

$$\emptyset.\texttt{file}(x) = M$$

$$\emptyset.\texttt{folder}(x) = M$$

$$(B - \{x\}).\texttt{file}(x) = M$$

$$(B - \{x\}).\texttt{file}(x/p) = M$$

$$(B - \{x\}).\texttt{folder}(x) = M$$

$$(B - \{x\}).\texttt{folder}(x/p) = M$$

$$(\texttt{folder} : x(B_1) \cup (B_2 - \{x\})).\texttt{file}(x) = M$$

$$(\texttt{file} : x(A) \cup (B - \{x\})).\texttt{folder}(x) = M$$

3.3 JavaCck class organization syntax

The JavaCck class organization syntax will be outlined in three 'rules' and a convention.

1. JCF's take the form of a hierarchical folder with files bearing the name of java classes (source class files), and java interfaces. The folders in a JCF are called packages.

2. JavaCck compliant file names have an extension `.java`. Files with a name carrying a `.java` extension are either source class files or interface files. Some mild restrictions on the use of identifiers for JavaCck were mentioned above.

3. A file carrying one ore more class sources or interface sources must carry the same name as one of the sources. Public classes must be placed in a file carrying the same name. Packages are stored in a directory carrying the name of the package. All files in a package `x` are preceded by a header: `"package x;"`.

4. An Output convention for JavaCck. The package `tjvap`[2] contains three classes: `bco` (boolean console output). `bco`, contains two methods: `t()` and `f()`; only by calling `bco.t()` and `bco.f()` can a JCF execution generate output. A call of `bco.t()` is executed by writing a line `true`,[3] a call of `bco.f()` by writing `false`. A modification of this class allows one to to write a boolean value. This class called `bcop` (for boolean console output with parameter), has a print method with a boolean parameter. The class

[2] Trivial JavaCck Void Action Package

[3] It is reasonable to assume these lines being written to the (hypothetical) screen containing the command line as well.

`cco` (for character console output) allows the printing of a few individual characters. All output instructions come in two versions, the one version generating a new line after each print instruction, the other generatien a white space instead.

3.4 The command environment

The command environment needed to experiment with JavaCck includes commands needed for editing JCF's as well as two commands taken from SUN's JDK:

- `javac s.java`, invoking the Java compiler on the start-up program. Execution of the compile command may lead to an error report on standard output; if it is empty, it expresses a successful completion of `javac`.
- `java s`, activating the Java Bytecode Virtual Machine (JBVM) on the compiler start-up program.

  The execution of `java s` will not enact any modifications on the JCF. However, it will produce an output file on standard error output. This file may either consist of a sequence of output messages or of some run-time error report generated by the JBVM machine in operation.

This text is entirely focussed on the behavior of these two commands, the editing of files being left to the reader's imagination.

Part II

Basic sequential features

Chapter 4

A JCF portfolio for BJC

The portfolio of this chapter is very small. It is based upon a start-up JCF called JCFm (m for minimal). All subsequent JCF's will be denoted by means of explicit FHN expressions. All JavaCck JCF's will contain the start-up class and one or more output classes[1]. They are first described in full detail.[2]
JCFm = `file:s.java`(

```
class s {
static void main (String x[ ]){c.m();}
}
```

) ∪ `file:bco.java`(

```
class bco {
static void t() {System.out.println("true");}
static void f() {System.out.println("false");}
}
```

)

4.1 Compilation and execution

4.1.1 Compiling JCFm

The java compiler can be applied to JCFm, by calling `javac` on `s.java`. The result is an error:

```
s.java:2: Undefined variable or class name: c
static void main (String x[ ]){c.m();}
```

Obviously JCFm is incomplete and the method call `c.m()` is at a loose end.

[1] The method `main` is called by the operating system in order to get an execution running. Its string array argument is used to accommodate command line inputs.

[2] Classes have so-called members. A classification of members is as follows: static value members, instance value members, constructors, static methods, and instance methods.

4.1.2 Completing JCFm

Two extensions for JCFm, JCFmt and JCFmf suffice to make it a JCF acceptable for the compiler.[3] The criterion for both extensions was to find the shortest possible classes the addition of which to JCFm leads to an observable difference in behavior.[4]

JCFmt = JCFm ∪ `file:c.java`(

```
class c {static void m(){bco.t();}}
```

)

JCFmf = JCFm ∪ `file:c.java`(

```
class c {static void m(){bco.f();}}
```

)

4.1.3 The effect of compilation on JCFmt and JCFmf

The results of compilation are denoted with JCFmtc and JCFmfc. These JCF's use named files with 'unreadable' contents (i.e. bytecodes), which will be denoted with $f1, \ldots, f6$ respectively.

JCFmtc = JCFmt ∪ `file:s.class`($f1$) ∪ `file:c.class`($f2$) ∪ `file:bco.class`($f3$), and
JCFmfc = JCFmf ∪ `file:s.class`($f4$) ∪ `file:c.class`($f5$) ∪ `file:bco.class`($f6$)
Some elementary exercises with the UNIX `diff` utility reveal that in this case the files $f1$ and $f4$ are identical as well as the files $f3$ and $f6$. The difference in generated compiler output is localized in the images of the differing classes only.

4.1.4 Running JCFmtc and JCFmtc

Running the programs by calling `java s` generates this output:
for JCFmt: `true`
for JCFmf: `false`

[3] The following JCF is accepted by the compiler as well, hence `main` need not be reserved for `s`. JCFmtmod1 = JCFm ∪ `file:c.java`(

```
class c {static void m(){d.main();}}
class d {static void main(){bco.t();}}
```

).

[4] It is a reasonable question to ask: find two java classes F and G of minimal length such that the classes can be embedded in identical contexts C[ ], giving rise to two Java programs with an observable difference in behavior.

4.2 The package tjvap

After the preceding preliminary explorations the accumulation of a portfolio of JCF's is now initialized from scratch, using an improved organization: output classes are collected in a package.

The package tjvap (JavaCck void action package) contains just one class: bco.java. Extensions of JavaCck will use extended packages. The following reorganization of the minimal TiviJava JCF is found:

JCFpa = file:s.java(

```
class s {
static void main (String x[ ]){c.m();}
}
```

) ∪ folder:tjvap(file:bco.java (

```
package tjvap;

public class bco {
public static void t() {System.out.println("true");}
public static void f() {System.out.println("false");}
}
```

))
The adapted version of JCFmt, JCFpat reads as follows: JCFpa ∪ (

```
import tjvap.*;
class c {static void m(){bco.t();}}
```

). Similarly an adapted version of JCFmf reads JCFpaf = JCFpa ∪ (

```
import tjvap.*;
class c {static void m(){bco.f();}}
```

)

4.3 The Publication Consistency Assumption

Access specifiers play an important role in the methodology of Java programming. By using access specifiers restrictions can be imposed on the use of values and methods from classes different from their origin class.

4.3.1 Access specifiers

Before providing additional examples we will give some information concerning access specifiers. Access specifiers are: 'default[5]', 'private', 'public' and 'protected'. These specifiers can be used as atributes of member declarations of

[5]The default occurs if no specifier is mentioned.

classes, a member being either a variable or a method. The specifiers determine restrictions on the access to class members from other classes. In most cases violations against these constraints will be found statically at compile time. In particular: private members cannot be accessed from any other class, public members can be accessed from all other classes. The other two options are in between, "default" being the more restrictive one. Protected members cannot be accessed from non-subclasses located in a different package, "default" members cannot be accessed from any class located in a different package. "no specifier" is also called: default protection. Private members cannot be accessed from other classes.

4.3.2 The public version operator

For a JCF K, its public version K_{public} is obtained by means of the following two steps:

1. make all classes public, which involves putting them in single class files,
2. turn all class member access specifiers into 'public' (this includes adding 'public' for default variables or methods.

PCA

PCA stands for Publication Consistency Assumption, expressing the following assertion:

If K is a JCF accepted by the java compiler and executes properly terminating with outputs α, then K_{public} is also accepted by the compiler and will equally execute with proper termination while producing outputs α.

Justification of PCA

The justification of PCA is primarily that it drasticallly simplifies the semantics of Java, making the operational semantics independent of access specifiers. PCA can be used as a 'design rule' to write programs in such a way that changing all access specifiers to `public` will not affect behavior. As a design rule it is perfectly usable and easily tested during design.

Nevertheless, PCA fails to hold for Java. As Bart Jacobs pointed out to us [8] some fairly elementary examples show the failure of PCA. A modified version of the examples can be foun in section 11.6. We have the impression that Java could be modified such as to satisfy PCA without losing any important properties. Therefore we propose to maintain PCA as a design rule.

4.3.3 Use of PCA

We will use PCA in the following way: access specifiers private and protected are hardly used in the sequel, the access specifier 'public' is of course used when necessary. This necessity occurs in the following cases: overriding of a member

of an interface in an implementing class, and reference to a method from a non-subclass contained in a different package (possible only if that member was declared public.)

The importance of access specifiers is in the possibility to force the compiler (type checker) to reject JCF's that may be problematic from a point of view of programming methodology. There is no connection between the use of access specifiers and the dynamic semantics of JCF's. Seen from the perspective of dynamic semantics, access specifiers are syntactic sugar only.

The dynamic semantics being our primary concern, we will ignore the type checking implications of access specifiers altogether.

4.4 Exploring nested method calls

Java allows, in principle, an unlimited nesting depth of method calls. In practice this depth will always be constrained by limitations of the implementing JBVM.

4.4.1 A well-founded recursion

JCF1 below compiles and executes without problem (computing true). JCF1 = JCFpa ∪ `file:c.java`(

```
class c1 {static void m1(){c2.m2();}}
class c {static void m(){c1.m1();}}
class c2 {static void m2(){c3.m3();}}
class c3 {static void m3(){c4.m4();}}
class c4 {static void m4(){c5.m5();}}
```

) ∪ `file:c5.java`(

```
import tjvap.*;
class c5 {static void m5(){bco.t();}}
```

)
It can be concluded that the order of classes within a file is not important.

4.4.2 Non-well-founded recursion

A mutual recursion will generate an infinite loop in principle but not in practice, the nested calls quickly generating a stack overflow in the execution JBVM. The conclusion is that BJC does not allow the writing of programs generating a diverging computation. An example is the JCF JCF2 = JCFpa ∪ `file:c.java`(

```
class c1 {static void m1(){c.m();}}
class c {static void m(){c1.m1();}}
```

) After some 70.000 method calls a stack overflow was reached. The error message in case reads as follows:

```
    1  Exception in thread "main" java.lang.StackOverflowError
    2          at c1.m1(Compiled Code)
    3          at c.m(Compiled Code)
    ........
    ........
74799          at c.m(Compiled Code)
74800          at c1.m1(Compiled Code)
74801          at c.m(Compiled Code)
74802          at s.main(Compiled Code)
```

turned out to have a considerable length. Redirecting the standard error output using (> &) to an ad hoc log file such long error messages can be made properly readable. The example can be simplified to JCF3 = JCFpa ∪ `file:c.java`(

```
class c {static void m(){m();}}
```

). Remarkably this simple program will run in 74.802 steps to its stack overflow just as well.

4.4.3 An output class using a toggle

The package `tjvap` is extended with a boolean console output class memorizing previous output. JCFpb = JCFpa ∪ `folder:tvjap(file:bcom.java`(

```
package tjvap;

public class bcom {
static boolean b = true;
public static void sw() {b = !b;}
public static void p() {if(b){System.out.println("true");}
                         else{System.out.println("false");}
                        }
}
```

). Let JCF4 = JCFpb ∪ `file:c.java`(

```
import tjvap.*;
class c {static void m(){bcom.p();c1.m1();}}
class c1 {static void m1(){bcom.sw();c.m();}}
```

). The execution of JCF4 (in JDK 1.1.6) generates this output followed by error message:

```
    1  true
    2  false
    3  true
    4  false
    5  true
 ....  ....
```

```
 5065  true
 5066  false
 5067  true
 5068  false
 5069  java.lang.StackOverflowError
 5070          at java.io.OutputStreamWriter.write(OutputStreamWriter.java)
 5071          at java.io.BufferedWriter.flushBuffer(BufferedWriter.java)
 5072          at java.io.PrintStream.write(PrintStream.java)
 5073          at java.io.PrintStream.print(PrintStream.java)
 5074          at java.io.PrintStream.println(PrintStream.java)
 5075          at tjvap.bcom.p(bcom.java:6)
 5076          at c.m(c.java:2)
 5077          at c1.m1(c.java:3)
 5078          at c.m(c.java:2)
 ...           ......
15210          at c.m(c.java:2)
15211          at c1.m1(c.java:3)
15212          at c.m(c.java:2)
15213          at s.main(s.java:2)
```

In JDK 1.2.1. we could only obtain a part of the output, not including the stackoverflow message.

4.4.4 Long computations

In order to find the practical limits of computation on our implementation we have written and run the following JCF. JCF5 = JCFpb ∪ `file:c.java`(

```
import tjvap.*;

final class c {
public static void m(){bcom.p();
                       m1();m1();m1();m1();m1();
                       m1();m1();m1();m1();m1();
                       bcom.p();
                       }
static void m1(){m2();m2();m2();m2();m2();
                 m2();m2();m2();m2();m2();}
static void m2(){m3();m3();m3();m3();m3();
                 m3();m3();m3();m3();m3();}
static void m3(){m4();m4();m4();m4();m4();
                 m4();m4();m4();m4();m4();}
static void m4(){m5();m5();m5();m5();m5();
                 m5();m5();m5();m5();m5();}
static void m5(){m6();m6();m6();m6();m6();
                 m6();m6();m6();m6();m6();}
static void m6(){m7();m7();m7();m7();m7();
```

```
                    m7();m7();m7();m7();m7();}
static void m7(){m8();m8();m8();m8();m8();
                    m8();m8();m8();m8();m8();}
static void m8(){bcom.sw();}
}
```

). This computation took some 8 minutes to terminate (fair enough for a 5-year-old machine). Completion produces output `true` twice.

Chapter 5

Extending the void action package

Some observations on packages will be made below. A complete coverage of meaningful cases is not attempted. As a first step the void action package is extended for subsequent use in the JCF portfolio for JavaCck.

Reduction of the operational capabilities of JavaCck to void output actions only introduces a significant simplification in relation to Java. This simplification enables one to formulate and pursue the aim of presenting a more or less complete picture of the language. Some flexibility is useful when giving more involved examples. We will provide some modifications of the output classes. In particular we will allow actions printing integers, the manipulation of integers by means of operations not being part of JavaCck.

5.1 Character output

In order to allow JCF's producing character console output a class is added to the output package. JCFpc = JCFpb ∪ `folder:tvjap(file:cco.java(`

```
package tjvap;

final public class cco {
public static void a() {System.out.println("a");}
public static void b() {System.out.println("b");}
public static void c() {System.out.println("c");}
public static void d() {System.out.println("d");}
public static void e() {System.out.println("e");}
public static void f() {System.out.println("f");}
public static void g() {System.out.println("g");}
public static void h() {System.out.println("h");}
```

```
public static void aw() {System.out.print("a ");}
public static void bw() {System.out.print("b ");}
public static void cw() {System.out.print("c ");}
public static void dw() {System.out.print("d ");}
public static void ew() {System.out.print("e ");}
public static void fw() {System.out.print("f ");}
public static void gw() {System.out.print("g ");}
public static void hw() {System.out.print("h ");}
}
```

). As an example of a computation consider: JCF6 = JCFpd ∪`file:c.java`(

```
import tjvap.*;
public class c {
public static void m(){
      bcom.p();cco.a();bcom.sw();cco.b();bcom.p();}
}
```

))
The output is:

```
true
a
b
false
```

5.2 Parametrized boolean output

To simplify the examples a further extension of the void action package is introduced. This class introduces two methods for printing a boolean argument. One method moves to the next line, the other method appends a white space. JCFpd = JCFpc ∪ `folder:tvjap(file:bcop.java`(

```
package tjvap;

final public class bcop {
public static void p(boolean q) {
     if(q){System.out.println("true");}
     else{System.out.println("false");}
     }
public static void pw(boolean q) {
     if(q){System.out.print("true ");}
     else{System.out.print("false ");}
     }
}
```

)).

5.3 Parametrized integer output

In some cases it is informative to use integer output. This allows one to easily trace the origin of a part of the printed output, the origin being the action of the program whose execution caused the printing of the integer. The package extends to JCFpe = JCFpd ∪ `folder:tvjap(file:ico.java(`

```
package tjvap;

public class ico {
public static void p(int i){System.out.println(i);}
public static void pw(int i){System.out.print(i + " " );}
}
```

)). As an example of a computation consider: JCF7 = JCFpe ∪`file:c.java(`

```
import tjvap.*;

class c {static void m(){
     int i = 1425,j = 0;cco.fw();bco.t();
     ico.p(354);ico.pw(25);ico.p(i);
     ico.pw(i);ico.p(j);
     }
}
```

). The output is:

```
f true
354
25 1425
1425 0
```

5.4 Packages in general

Packages are a non-trivial modularization mechanism. Some additional examples are necessary to demonstrate the possibilities of packages. Without some significant programming environment support, working with nested packages is rather unattractive, however. The package-naming conventions are part of JavaCck class organization syntax, identical to the conventions for Java. Our style of presentation turns out to be ineffective for the notational complexity of deeper package hierarchies. Because it is impossible to present these in FHN in a way that is comprehensible for the human eye, we have omitted such examples.

As packages are not absolutely essential for object-oriented programming we have taken the liberty not to study them in detail.

Chapter 6

Local and static boolean variables

This chapter extends the JCF portfolio with examples of the use of boolean variables, in particular local variables and static variables.

The preceding examples exist in BJC. The next sequence of examples illustrates the behavior of JCF's using a combination of the following features: BJC-features, local boolean variables, boolean fields, boolean assignments.

The examples concentrate mainly on static variables, the variables local to a method body not presenting any unusual characteristics.

6.1 Local and static boolean variables

A number of JCF's exemplify available options. The following example illustrates a number of ways local and static boolean variables can be used. JCF8 = JCFpd ∪ `file:c.java` (

```
import tjvap.*;

class c {
static void m() {
      boolean b = true;bcop.p(b);
      boolean c;bcop.pw(true || c);bcop.p(false && c);
      boolean d = true || c;bcop.pw(d);d = !d;bcop.p(d);
      bcop.pw(c1.e);bcop.pw(c1.f);bcop.p(c1.g);
      c1.gneg();bcop.p(c1.g);
      c1.g = true;c1.f = false;bcop.pw(c1.f);bcop.p(c1.g);
      c1.g2f();bcop.p(!!c1.f);
      }
}
```

```
class c1 {
static boolean e = true;
static boolean f = !e;
static boolean g = e && (f || !e);
static void gneg() {g = !g;}
static void g2f() {f = g;}
}
```

) JCF8 computes outputs:

```
true
true false
true false
true false false
true
false true
true
```

The next example illustrates the presence of several methods inside a class. JCF9 = JCFpd ∪ `file:c.java` (

```
import tjvap.*;

class c {
static boolean b = true, e = true;
static void m() {bcop.pw(true);bcop.pw(b);bcop.p(!e);
                 m1();bcop.p(b);
                 c.m1();bcop.p(b);
                 e = e && b;bcop.p(!(b || e));
                }
static void m1() {b = !e;}
}
```

) This JCF computes outputs:

```
true true false
false
false
true
```

The following example regards once more the use of static class variables from another class. JCF10 = JCFpd ∪ `file:c.java` (

```
import tjvap.*;

class c {
static boolean b = true;
static void m(){
      bcop.p(b);
```

```
      bcop.p(c1.b);
      bcop.p(c1.c);
      c1.b = false;bcop.p(c1.b);
      c1.m1();bcop.p(c1.d);
      c1.m2(b,c1.b);
      bcop.p(c1.z);
      }
}

class c1 {
static boolean x = false;
static boolean z = false;
static boolean b = !x;
static boolean c = false;
static boolean d = b && !c;
static void m1(){b = !d;}
static void m2(boolean x,boolean y){
      z = (x && c) || (b || y);}
}
```

) JCF10 computes outputs:

```
true
true
false
false
true
false
```

6.2 Initialition of static variables

Important aspects of Java variable handling can be studied in connection with static variable initialization. A number of cases will be considered.

6.2.1 Default initialization to false

Its is an important design decision for Java, inherited from C, to initialize boolean variables not given an explicit initialization in a JCF with a default boolean value. JCF11 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

final class c {
static boolean b;
public static void m(){bcop.p(b);}
}
```

). The static variable gets a meaningful default initial value: `false` which is printed upon execution of JCF11. In JCF12 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

final class c {
static boolean b = true;
public static void m(){bcop.p(b);}
}
```

) the initial assignment overwrites the default assignment and output `true` is produced.

6.2.2 Inconsistent initializations are accepted

An interesting example is JCF13 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

final class c {
static boolean b1 = d.b2;
public static void m(){bcop.p(b1);bcop.p(d.b2);d.m();}
}

final class d {
static boolean b2 = !c.b1;
public static void m(){bcop.p(b2);}
}
```

), which produces three times `true`. Changing the order of class presentation makes no difference to the output.

6.2.3 Declaration before use

In JCF14 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

final class c {
static boolean b1;
static boolean b2 = !b1;
public static void m(){bcop.p(b1);bcop.p(b2);}
}
```

) the outputs are `false` followed by `true`. Changing the order of the declarations causes the compiler to complain about a forward reference to a variable. Clearly, inside the declarations of static variables, declaration must precede use. As soon as the compiler accepts a declaration sequence, its order within the sequence of declarations is immaterial however (i.e. legal changes in declaration order lead to identical functionalities).

6.3 Symmetry-breaking initialization mechanisms

Not all system behavior involving static class variables is easy to predict. This extended section covers one example of non-trivial behavior by considering several related examples. The methodological conclusion is that whenever static class variables are initialized across classes, empirical semantics is very useful in analysing the working of a program.[1]

6.3.1 A symmetric JCF with an asymmetric result

We will consider a form of JCF having two output instructions and admitting a name permutation/class permutation which results in a permutation of the two output intructions. Remarkably this symmetry should leave the output invariant. Consider: JCF15 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

final class c {
public static void m(){
     bcop.p(d1.b1); // (first output instruction)
     bcop.p(d2.b2); // (second output instruction)
     }
}

final class d1 {
static boolean b1 = !d2.b2;
}

final class d2 {
static boolean b2 = !d1.b1;
}
```

), the output is `false`, `true`. If we modifiy the second output instruction of JCF15 so as to become a repetition of the first output instruction the output changes to `false`, `false`. The first static boolean used in actual execution is given a default value `false`, the second value's initialization is fully determined. Therefore, interchanging the declaration order of the classes `d1` and `d2` will not change the output and neither will changing the order of the output statements in the method of `c`.

[1] It is very well possible, though at the level of empirical semantics not decidable, that the behavior of JavaCck programs with initialization cycles is completely implementation dependent. That happens if the language definition gives no information in such cases. When programming in practice, it is important to use Java in such a way that the official language definition makes the output predictable sufficiently well to guarantee some relevant program behavior.

6.3.2 Changing the order of outputs

However, by having the output instructions preceded with a seemingly futile use of `b2` a difference appears. JCF16 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

final class c {
public static void m(){
     boolean b = d2.b2; // (relevant use of d2.b2)
     bcop.p(d1.b1); // (first output instruction)
     bcop.p(d2.b2); // (second output instruction)
     }
}

final class d1 {
static boolean b1 = !d2.b2;
}

final class d2 {
static boolean b2 = !d1.b1;
}
```

) produces the output `true, false` differing from the output generated by JCF15. Similarly JCF17 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

final class c {
public static void m(){
     d2.b2 = d2.b2; // (relevant use of d2.b2)
     bcop.p(d1.b1); // (first output instruction)
     bcop.p(d2.b2); // (second output instruction)
     }
}

final class d1 {
static boolean b1 = !d2.b2;
}

final class d2 {
static boolean b2 = !d1.b1;
}
```

) produces `true, false`.

Chapter 7

The conditional construct and iteration

This section contains a brief excursion to the class notation extending BJC with boolean fields, local boolean variables, boolean assignments, the while-loop, the conditional construct, the return statement and boolean-returning static methods.

7.1 Boolean-returning static methods

First the use of boolean-returning static methods (bRCM) and the corresponding return statement RET is dealt with.

The following JCF demonstrates a boolean-returning method (preceded by the keyword boolean instead of the keyword void). The body of the method should end (or rather terminate) with the execution of a return statement which produces the boolean value that will be returned.

JCF18 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
public static void m(){
     bcop.pw(d1.b1());
     bcop.pw(d2.b2);
     bcop.p(d2.c2());
     }
}

class d1 {
static boolean b1() {return true;}
static boolean b1(boolean x) {return x || !b1();}
```

```
}

class d2 {
static boolean b2 = d1.b1(false);
static boolean c2() {b2 = true;return d1.b1(b2);}
}
```

) produces `true false true`.

7.1.1 RET for booleans, a second example

The following example regards the basics of returning a boolean value. JCF19 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
      static void m() {
            bcop.p(f());
            bcop.p(g());
            bcop.p(h());
            bcop.p(i());
            bcop.p(j());
      }
      static boolean b = true;
      static boolean f() {
            if(true){return true;};cco.aw();return false;}
      static boolean g() {
            if(true){return b;};cco.bw();return !b;}
      static boolean h() {cco.cw();return !b;}
      static boolean i() {cco.dw();
            {boolean b = false;cco.ew();return !b;}
            }
      static boolean j() {{{{return false;};};;;;
            ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
            };{};{};{};{};{};{};{};{};{};{};{};{};{}
            {};;;;;}
}
```

). It produces

```
true
true
c false
d e true
false
```

One is tempted to omit the conditionals in this case (replacing `if(true){return true;}` by `return true;`). The compiler will not admit this, however, complaining that instructions directly following `return true;` cannot be reached.[1]

7.2 Conditional programs

The conditional construct allows a program to go either way depending on a test. A systematic example is contained in JCF20 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {static void m(){
      if(true){cco.aw();}
      if(false){cco.bw();};bco.t();
      if(true){cco.aw();}else{cco.bw();}
      if(false){cco.dw();}else{cco.ew();};bco.f();
      boolean b1 = true;
      if(b1){cco.aw();}
      if(!b1){cco.aw();}else{cco.bw();}bco.t();
      if(c1.x1()){cco.cw();}else{cco.dw();}bco.f();
      if(c1.x2()){cco.cw();}else{cco.dw();}bco.t();
      if(c1.x1()||c1.x2()){cco.cw();}else{cco.dw();}bco.f();
      }
}

class c1 {
      static boolean x1() {cco.aw();
            if(true){return false;}else{return true;};}
      static boolean x2() {cco.bw();
            if(x1()){return !x1();} else {return true;};}
}
```

) which produces

```
a true
a e false
a b true
a d false
b a c true
a b a c false
```

[1] It is remarkable that Java defeats the transformation `if(true){X} = X`.

7.3 Iteration

There are several iteration constructs in Java of which JavaCck contains only one: the while-construct. The following example documents its functionality. JCF21 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {static void m(){boolean b0 = false;
      while(b0){cco.aw();}
      while(!b0){cco.bw();b0 = !b0;}
      b0 = !b0;
      while(false){};
      bco.t();
      boolean b1 = true;
      while(!b0 && b1){while(!b0){b0 = true;cco.ew();};
                       while(b0){b0 = false;while(b1){b1 = false;}}}

      while(c1.x1()){cco.cw();};{cco.dw();}bco.f();
      if(c1.x2(true)){cco.cw();}else{cco.dw();}bco.t();
      if(c1.x4||c1.x2(b0)){cco.cw();}else{cco.dw();}bco.f();
      }
}

class c1 {
      static boolean x1() {cco.aw();return false;}
      static boolean x2(boolean b) {cco.bw();
            while(x1()){return !x1();}; return true;}
      static boolean x3 = x1();
      static boolean x4 = x2(x3);
}
```

) producing

```
b true
e a b a a d false
b a c true
c false
```

7.3.1 Diverging computations

In contrast to recursion, the while loop can in principle give rise to a diverging computation, as it needs not create an ever-growing stack inside the executing JBVM. JCF22 = JCFpd ∪ `file:c.java` (

```
import tjvap.*;
```

```
class c {
static boolean b = false;
static void m(){
      while(b || !b){};
      cco.a();
      }
}
```

)
Computation of this JCF fails to terminate. The seemingly needless complexity of the body is used to prevent the compiler from finding out statically that the loop cannot terminate.

7.3.2 Iteration guards

The connection between conditions and while loops or conditional constructs merits some further attention. A brief excursion to empirical syntax reveals why some examples of infinite loops become slightly more complex than expected. Consider the following error report from the Java compiler: let JCF23 = JCFpd ∪ `file:c.java` (

```
import tjvap.*;

public class c {
public static void m(){while(true){};bco.t();}
}
```

), its compilation leads to:

```
c.java:4: Statement not reached.
      public static void m(){while(true){};bco.t();}
```

A similar error is reported for JCF24 = JCFpd ∪ `file:c.java` (

```
import tjvap.*;

public class c {
public static void m(){while(false){bco.t();}}
}
```

). Remarkably the compiler will not signal the existence of an unreachable statement in JCF25 = JCFpd ∪ `file:c.java` (

```
import tjvap.*;

public class c {
public static void m(){if(false){bco.t();}}
}
```

).

Chapter 8

Four-valued logic

This chapter describes the effect of logical computations with booleans that are computed by means of static methods. A four-valued logic is helpful for a systematic understanding of the logical operators in this case. We will use the notation of [2] to discuss this particular 4-valued logic. A boolean-returning static method may return `true` (T), or `false` (F). If neither is the case, a run-time error may be reported (erroneous termination represented by meaningless M) or the computation may be diverging (D). The connectives '||' (disjunction) and '&&' (conjunction) are left-to-right operators. First the left-hand argument is evaluated. Then, if possible, an output is produced. Otherwise the right-hand argument is evaluated as well. This order of events explains the error propagation observed.

8.1 Four-Valued Logic

In this section the description of 4-valued logic from [2] is reproduced in the formulation of [4]. This logic underlies the phenomena of Java an many other program notations. The relation between Java propositional operators and the ones below is as follows: $x \,\&\&\, y = x \mathbin{\wedge\!\!\!\circ} y$ and $x \,||\, y = x \mathbin{\vee\!\!\!\circ} y$ with $x \mathbin{\vee\!\!\!\circ} y = \neg(\neg x \mathbin{\wedge\!\!\!\circ} \neg y)$. The logic is based on the set of values $\mathbb{T}_4$, which consists of

- M (meaningless),
- T (true),
- F (false), and
- D (divergence).

The following set of logical operations is defined and distinguished as truth-functionally complete:

- $\neg$ (negation),
- $\downarrow$ (definedness: distinguishing F, T from D, M),
- $\wedge$ (conjunction), and
- $\wedge\!\!\!\circ$ (left-sequential conjunction).

Here $\not\wedge$ denotes McCarthy's left-to-right conjunction, where we adopt the asymmetric notation from [2]. Truth tables for $\neg, \downarrow, \wedge$, and $\not\wedge$ are

x	$\neg x$
M	M
T	F
F	T
D	D

x	$\downarrow x$
M	F
T	T
F	T
D	F

$\wedge$	M	T	F	D
M	M	M	M	M
T	M	T	F	D
F	M	F	F	F
D	M	D	F	D

$\not\wedge$	M	T	F	D
M	M	M	M	M
T	M	T	F	D
F	F	F	F	F
D	D	D	D	D

(Observe that $\wedge$ is monotonic in M.) The resulting logic is denoted as

$$\Sigma_4(\neg, \downarrow, \wedge, \not\wedge),$$

where the subscript 4 refers to the four values of $\mathbb{T}_4$. Notice that the connectives $\wedge$, $\not\wedge$, and their duals are associative, and that $\wedge$ and $\vee$ are commutative as well.[1]

8.2 Left-most inner-most evaluation for booleans I

The following example concerns the evaluation order (left to right) for the boolean connectives. This order matters first of all in case the booleans are to be computed by means of boolean-returning methods.

The correspondence with the truth-tables is 4-valued logic is completely satisfactory. JCF26 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
      static void m() {
            bcop.pw(true || f());
            bcop.pw(false && f());
            bcop.pw(false && g());
            bcop.pw(true && (h() || f()));
            bcop.pw(false || f());
      }
      static boolean f() {return f();}
      static boolean g() {boolean x = true;
            while(x || !x){};return true;}
      static boolean h() {return true;};
}
```

), its execution leads to:

[1] In connection with 4-valued logic the following facts are of relevance: Kleene's 3-valued logic is embedded in $\Sigma_4(\neg, \downarrow, \wedge, \not\wedge)$ by $\mathsf{T}, \mathsf{F}, \mathsf{D}$ and $\neg, \wedge$, Bochvar's strict 3-valued logic by $\mathsf{T}, \mathsf{F}, \mathsf{M}$ and $\neg, \wedge$, and McCarthy's sequential 3-valued logic by T, F, one of D, M, and $\neg$, $\not\wedge$, where the asymmetric symbol for left-sequential conjunction is introduced in [2].

```
     1  true
     2  false
     3  false
     4  true
     5  Exception in thread "main" java.lang.StackOverflowError
     6          at c.f(Compiled Code)
     7          at c.f(Compiled Code)
     8          at c.f(Compiled Code)
.....................................
698120          at c.m(Compiled Code)
698121          at s.main(Compiled Code)
```

It can be concluded that an argument which will diverge or lead to an error will not propagate its divergence or error if the argument need not be evaluated.

8.3 Left-most inner-most evaluation for booleans II

If the subexpressions of a boolean expression consist of static fields, even more interesting initialization sequences can be observed. In this case the correspondence with the truth-table for 4-valued logic is also unproblematic. In the following example the logical connectives are alternatively wrapped in a new method. The result is that this method will enforce evaluation of the arguments before execution of the body. Unlike in the case of the built-in boolean connectives for conjunction and disjunction both arguments will be evaluated, the connection with the 4-valued logic getting lost as a consequence. JCF27 = JCFpd ∪ `file:c.java` (

```
import tjvap.*;

class c {
static boolean or(boolean x,boolean y) {return x || y;}
static boolean and(boolean x,boolean y) {return x && y;}
      static void m() {
            bcop.pw(true || H1.f);bco.t();
            bcop.pw(true && H1.f);bco.f();
            bcop.pw(true && H1.f);bco.t();
            bcop.pw(false && H2.g);bco.f();
            bcop.pw(H2.g && H1.f);bco.t();
            bcop.pw(true && (H2.h1 || false));bco.f();

            bcop.pw(or(true,H1.f()));bco.t();
            bcop.pw(and(true,H1.f()));bco.f();
            bcop.pw(or(false,H1.f()));bco.t();
            bcop.pw(and(false,H2.g()));bco.f();
```

```
            bcop.pw(and(H2.g(),H1.f()));bco.t();
            bcop.pw(and(false,H2.h1 || H4.j()));bco.f();
      }
}

class H1 {
      static boolean f = f();
      static boolean f() {cco.aw();return true;}
}

class H2 {
      static boolean g = g();
      static boolean g() {cco.bw();return false;}
      static boolean h = !H3.i;
      static boolean h1 = h1();
      static boolean h1() {return H4.j;};
}

class H3 {
      static boolean i = i();
      static boolean i() {cco.cw();return true;}
}

class H4 {
      static boolean j = j();
      static boolean j() {cco.dw();return false;}
}
```

), its execution leads to the following sequence of outputs:

```
true true
a true false
true true
false false
b c d false true
false false
a true true
a true false
a true true
b false false
b a false true
d false false
```

8.4 Boolean valued methods with side effects

The 4-valued logic gives a simplified picture in some cases. Not all expressions of type boolean can be interpreted as expressions for this 4-valued logic. The complications arise in the case of side effects. Consider JCF28 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
      static void m() {
            bcop.p(f() || !f());
            bcop.p(!f() && f());
      }
      static boolean b = true;
      static boolean f() {b = !b;return b;}
}
```

), its execution leads to:

```
false
true
```

8.5 Scopes for boolean fields and methods

The flexibility of notation offered by Java is illustrated by the next JCF: JCF29 = JCFpd ∪ `file:c.java` (

```
import tjvap.*;

class c {
      static boolean c = false;
      static boolean b() {return true;}
      static boolean b(boolean b) {return b;}
      static void m() {
            bcop.pw(c);bcop.pw(b());bcop.pw(b(false));
              cco.a();
            bcop.pw(b(true));bcop.pw(b.b);bcop.pw(b.c);
              cco.b();
            bcop.pw(b.b());bcop.pw(b.b(false));bcop.pw(b.b(b()));
              cco.c();
            bcop.pw(b.b(b.b));bcop.pw(b(!c));bcop.pw(!c);
              cco.d();
            }
}
```

```
class b {
      static boolean b = true;
      static boolean c = true;
      static boolean b() {return b;}
      static boolean b(boolean b) {return !b;}
}
```

), its execution leads to:

```
false true false a
true true true b
true true false c
false true true d
```

Part III

Objects and inheritance

Chapter 9

Class instantiation: objects

This section contains a sequence of examples of computations involving class instances, also called objects. All objects are class instances. Object creation in a program is recognizable: the keyword `new` is an indication of an instruction or a part of an instruction creating an object.

It is tempting to try and provide a definition of an object in advance. However, it will prove very difficult to produce such a definition. The collection of examples below should produce a clear intuition concerning the notion of an object in (Trivi)Java.[1] An object is an abstract item, which can be used to model the effect of object-creating instructions in object-based program notations. It is easy and conceptually correct to think of an object as a time-stamped sequence number of an act of creation, together with a 'sufficient amount of bookkeeping' regarding that act of creation. Names in Java can be used to 'talk about' objects but should not be confused with the objects themselves. Java has no direct notation for objects. The pointers of C and C^{++} are much closer to objects than anything in Java. Java contains the `equals(-)` method, allowing a test on an object revealing whether that object 'equals' another one. With considerable care and limitations objects can be viewed as 'equals'-equivalence classes. 'Equals'-equivalent objects can show different behavior, however, making the intuitions far from trivial after all.

9.1 Terminology

Some terminology is needed. All terminology is to be used relative to some JCF. In a JCF there may be some classes. Static fields associate values or objects with the classes whereas instance fields associate values or objects with instances of classes. Static fields are also called: class fields, static values, static value members, value class members. Methods are either attached to classes or to objects (class instances). A method connected with a class is called a class

[1] These examples are not meant to illustrate the usage of objects in professional Java programming.

method or static method. Such methods are also called static members of the class or even class members of the class. In the case a method is attached to an object it is called an instance method or non-static method. Instance methods are among the non-static members of a class.

Variables are either local names within a method body or names of static fields or of instance fields. Method parameters also play the role of variables. (The scoping rules are not easy to grasp.)

9.2 Objects with boolean instance fields

The simplest cases of object instantiation and usage involve instance fields only. JCF30 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
public static void m(){
      c1 x = new c1();
      bcop.pw(c1.d);bcop.pw(c1.e);bcop.pw(x.b);bcop.p(x.c);
      }
}

class c1 {
      boolean b = true;
      boolean c = false;
      static boolean d = true;
      static boolean e = false;
}
```

). This JCF produces `true false true false`.

9.2.1 Assignments to instance fields

Assignments are possible on instance fields, as well as on class fields: JCF31 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
public static void m(){
      c1 x = new c1();
      c1.d = false;c1.e = true;x.b = false;x.c = true;
      bcop.pw(c1.d);bcop.pw(c1.e);bcop.pw(x.b);bcop.pw(x.c);
      cco.a();
      x.b = x.c;
      c1 y = new c1();
```

```
      y.b = y.c = x.c = false;
      bcop.pw(y.b);bcop.pw(y.c);bcop.pw(x.c);cco.a();
      bcop.pw(x.b);bcop.pw(y.d);cco.b();
      }
}

class c1 {
      boolean b = true, c = false;
      static boolean d = true, e = false;
}
```

). The generated output is

```
false true false true a
false false false a
true false b
```

9.2.2 Accessing static fields via an object

The static fields of a class can be acccessed via a selector on the class (the field name serving as the selector), and also via selection on instances of the class. JCF32 = JCFpe ∪ `file:c.java`(

```
import tjvap.*;

public class c {
public static void m(){
      c1 x = new c1();
      x.d = true;
      bcop.pw(x.d);bcop.pw(c1.d);ico.p(1);
      x.d = false;
      bcop.pw(x.d);bcop.pw(c1.d);ico.p(2);
      c1.d = true;
      bcop.pw(x.d);bcop.pw(c1.d);ico.p(3);
      x.e = true;
      bcop.pw(x.e);bcop.pw(c1.e);ico.p(4);
      }
}

class c1 {
      static boolean d = true;
      static boolean e = false;
}
```

). The generated output is

```
true true 1
false false 2
```

```
true true 3
true true 4
```

9.2.3 Definition before use: not always!

For static fields definition before use is required. For instance fields just as well. The use of a static field for the initialization of an instance field may, however, precede the definition of the static field. JCF33 = JCFpe ∪ `file:c.java`(

```
import tjvap.*;

public class c {
public static void m(){
      c1 x = new c1();
      bcop.pw(x.a);
      x.a = true;
      bcop.pw(x.a);ico.p(1);
      bcop.pw(c1.e);bcop.pw(c1.d);ico.p(2);
      }
}

class c1 {
      static boolean d = true;
      boolean b = e;
      boolean a = b;
      static boolean e = !d;
}
```

). The generated output is

```
false true 1
false true 2
```

The definition before use rule seems to be violated in the next JCF, the indirect acces to a static variable via an object solving the problems unexpectedly. JCF34 = JCFpe ∪ `file:c.java`(

```
import tjvap.*;

public class c {
public static void m(){
      bcop.pw(c1.a);
      bcop.pw(c1.b);
      bcop.pw(c1.x.a);
      bcop.p(c1.x.b);
      }
}
```

```
class c1 {
      static c1 x = new c1();
      static boolean b = !x.a;
      static boolean a = false;
}
```

). The generated output is

```
false true false true
```

9.3 Objects with instance methods

The next sequence of examples concerns the instantiation of classes in the presence of instance methods. Special attention is paid to cumulative method application.

9.3.1 Application to a local object

A quite minimal example of the use of instantiation and instance methods is this: JCF35 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
static c1 x = new c1();
static void m(){x.m();}
}

class c1 {void m(){cco.a();}}
```

). The result of execution of JCF35 consists of the production of an '`a`'. In this case an instance method has been applied to a locally defined object. The object plays the role of the class name.

9.3.2 Application of static methods to a local object

In the next JCF a static method is applied to an object. JCF36 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
static c1 x = new c1();
static void m(){x.m1();}
}

class c1 {static void m1(){cco.a();}}
```

). The output of the computation is `a`, indicating the role of the object name as a mere placeholder for its class. The conclusion is that instance methods can only be applied to instance whereas static methods are applicable to classes and class instances alike.

9.3.3 Instance method application to a non-local object

The object can be defined in another class as well: JCF37 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
public static void m(){c1.x.m();}
}

class c1 {
static c1 x = new c1();
public void m(){cco.a();}
}
```

) will produce the same output. It can also be defined in its own class. JCF38 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
public static void m(){c2.x.m();}
}

class c1 {
public void m(){cco.a();}
}

class c2 {
static c1 x = new c1();
}
```

) produces the same output once more.

9.3.4 Repeated member selection

An unlimited degree of indirection is possible, the JCF below producing 'a' twice: JCF39 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
public static void m(){
```

```
      c3.x.y.y.m();
      (((c3.x).y).y).m();
      }
}

class c0 {
public void m(){cco.a();}
}

class c1 {
c0 y = new c0();
}

class c2 {
c1 y = new c1();
}

class c3 {
static c2 x = new c2();
}
```

). The additional brackets of the second print instruction are permissible but redundant. They just assert the default execution order of method call expressions.

9.3.5 Repeated method application: inner-most evaluation

Further complexity can be introduced by using non-void methods: JCF40 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
public static void m(){
      c3.x.f().f().f().y.g().g().y.m();
      c3.x.y.h().x.y.h().x.y.h().x.f().y.y.m();
      }
}

class c0 {
public void m(){cco.a();}
}

class c1 {
c0 y = new c0();
c1 g() {cco.cw();return this;}
```

```
c3 h() {cco.dw();c3 z = new c3();return z;}
}

class c2 {
c1 y = new c1();
c2 f() {cco.bw();return this;}
}

class c3 {
static c2 x = new c2();
}
```

) which computes

```
b b b c c a
d d d b a
```

These examples suffice to illustrate the inner-most evaluation strategy of the JBVM.[2]

9.3.6 Left-most inner-most evaluation

Yet another modification of these examples illustrates the left-most inner-most evaluation strategy for Java expressions. JCF41 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
public static void m(){
      c1.f(c4.x.g2().g2(),c4.x.g1()).y.m();
      c2.f(c4.x.g1().g1(),c4.x.g2()).y.m();
      }
}

class c0 {
public void m(){cco.a();}
}

class c1 {
static c3 f(c3 x,c3 y) {cco.cw();return x;}
}

class c2 {
static c3 f(c3 x,c3 y) {cco.bw();return x;}
}
```

[2]Innermost evaluation refers to the structure of expressions. In order to see that, structure describing brackets have to be added in most cases.

```
class c3 {
c0 y = new c0();
c3 g1() {cco.dw();return this;}
c3 g2() {cco.ew();return this;}
}

class c4 {
static c3 x = new c3();
}
```

) which computes

```
e e d c a
d d e b a
```

9.4 Overloading

Overloading takes place if different methods can only be distinguished on the basis of their argument types. An example is JCF42 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
static void f(c1 x) {cco.aw();}
static void f(c2 x) {cco.bw();}
static void f(c3 x) {cco.cw();}
static void m(){
     c1 x1 = new c1();
     c2 x2 = new c2();
     c3 x3 = new c3();
     f(x1);f(x2);f(x3);bco.t();
     }
}

class c1 { }
class c2 { }
class c3 { }
```

) which computes

```
a b c true
```

Innermost evaluation is important for overloading as well. JCF43 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;
```

```
class c {
static c2 g(c1 x) {cco.aw();return new c2();}
static c3 g(c2 x) {cco.bw();return new c3();}
static c1 g(c3 x) {cco.cw();return new c1();}
static void m(){
     c1 x1 = new c1();
     c2 x2 = new c2();
     c3 x3 = new c3();
     g(x1);g(x2);g(x3);bco.t();
     g(g(x1));g(g(x2));g(g(x3));bco.f();
     g(g(g(x1)));g(g(g(x2)));g(g(g(x3)));bco.t();
     }
}

class c1 { }
class c2 { }
class c3 { }
```

) which computes

```
a b c true
a b b c c a false
a b c b c a c a b true
```

9.5 Objects with instance fields of their own class

Class instances may have value members of their own class. The syntax will admit this. Operationally it leads to unavoidable errors in case such members are initialized.

9.5.1 Field initialization errors

If a class has (owns) a value member of the same type, each attempt to create an instance for the class will recursively invoke yet another attempt to create an object. This kind of dynamic error will always occur if the class/value member hierarchy features cycles. JCF44 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
public static void m(){c2.x.m();}
}

class c1 {
c1 y = new c1();
public void m(){cco.a();}
```

```
}

class c2 {
static c1 x = new c1();
}
```

) produces the following error

```
   1  java.lang.StackOverflowError
   2          at c1.<init>(c.java:8)
 ..................................
7800          at c1.<init>(c.java:8)
7801          at
7802          at
7803          at s.main(s.java:2)
```

A stack overflow may emerge in the absence of any method calls in the bodies of classes in a JCF: JCF45 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
static c x = new c();
c y = new c();
static void m(){ }
}
```

). JCF45 is accepted by the compiler, its execution giving rise to an immediate stack overflow.

9.5.2 Non-initialized instance variables

The stack overflow in example JCF39 can be removed by dropping the initialization of the 'dangerous' instance variable: JCF46 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
public static void m(){c2.x.m();}
}

class c1 {
c1 y;
public void m(){cco.a();}
}

class c2 {
static c1 x = new c1();
}
```

) computes 'a'.

9.5.3 Owner class yielding methods

It is unproblematic for class instances to have operations on them that yield objects of the same type, the difference being that these operations need not immediately be computed if an object is constructed. JCF47 = JCFpd ∪ file:c.java(

```
import tjvap.*;

public class c {
public static void m(){c1.z.y(c2.x).m();}
}

class c1 {
static c1 z = new c1();
c1 y(c1 z) {return new c1();}
public void m(){cco.a();}
}

class c2 {
static c1 x = new c1();
}
```

) will compute 'a'.

9.6 Constructors

A constructor is a non-static method of a class carrying the exact name of the class, no target type being specified. Constructors may have arguments. It is understood that every class definition introduces by default a constructor without parameters. If other constructors are introduced the default constructor ceases to exist. The body of a constructor should not return a value. (More precisely: it should not use the `return` instruction.)

Constructors can be considered syntactic sugar because constructors can always be replaced by explicit methods. The use of constructors simplifies class writing, however. Moreover, there is an important interaction between written constructors and the default constructor prevailing in the absence of any written constructors.

9.7 Constructors by example

A very simple use of a constructor is found in JCF48 = JCFpd ∪ file:c.java(

```
import tjvap.*;

public class c {static void m() {c1 x = new c1();}}

class c1 {c1() {cco.a();}}
```

) which computes 'a'. The application of this constructor, overriding the default vonstructor, is a side effect to print an 'a'.

9.7.1 A constructor updating a static field

In the following case the constructor changes a static field. JCF49 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {static void m() {
      bcop.pw(c1.b);c1 x = new c1();bcop.p(c1.b);}
}

class c1 {static boolean b = false;
      c1() {cco.aw();b = true;}
}
```

) which computes '`false a true`'.

9.7.2 A constructor updating an instance field

Constructors can affect instance fields just as well. JCF50 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {static void m() {
      c1 x = new c1();bcop.p(x.b);}
}

class c1 {boolean b = false;
      c1() {cco.aw();b = true;}
}
```

) computes '`a true`'.

9.7.3 Constructors with parameters

The following JCF exemplifies a characteristic use of parametrized constructors: JCF51 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {static void m() {
      c1 x = new c1(true, false);bcop.pw(x.b);bcop.p(x.c);
      c1 y = new c1(true, true);bcop.pw(y.b);bcop.p(y.c);
      c1 z = new c1(false, true);bcop.pw(z.b);bcop.p(z.c);
      c1 u = new c1(false, false);bcop.pw(u.b);bcop.p(u.c);
      x = z;
      bcop.pw(x.b);bcop.p(x.c);
      z.b = true;
      bcop.p(x.b);
      }
}

class c1 {boolean b = false, c = true;
      c1(boolean _b, boolean _c) {
            b = _b;c = _c;}
}
```

) which computes

```
true false
true true
false true
false false
false true
true
```

The previous examples also illustrates the effect of assignments across constructed objects.

9.7.4 Constructors and static variable initialization

Initializations come in a certain order. This can be visualized in cases involving constructors. A very simple use of constructors is JCF52 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
      static c1 x1 = new c1();
      static c2 x2 = new c2();
      static void m() {
            c3 x3 = new c3();bcop.pw(true);
            c4 x4 = new c4();bco.f();}
}
```

```
class c1 {c1() {cco.aw();}}
class c2 {c2() {cco.bw();}}
class c3 {c3() {cco.cw();}}
class c4 {
      static c2 x2 = new c2();
      static c1 x1 = new c1();
      c4() {cco.dw();c3 x3 = new c3();}
}
class c5 {c5() {cco.ew();}}
```

) which computes '`a b c true b a d c false`'.

9.7.5 Overloaded constructors

Another example illustrates the fact that constructors may use parameters. In addition it illustrates that constructors can be overloaded. JCF53 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
      static c1 x1 = new c1();
      static c2 x2 = new c2();
      static c1 x1b = new c1(true);
      static c1 x1c = new c1(!c2.b);
      static c2 x2b = new c2();
      static void m() {bcop.pw(c2.b);c2.b = !c2.b;
            cco.aw();
            x1.n();x2.n();x1b.n();x1c.n();x2b.n();
            cco.c();
            }
}

class c1 {
      c1() {cco.cw();bcop.pw(c2.b);}
      c1(boolean b) {cco.cw();bcop.pw(b);}
      void n() { }
}

class c2 {
      c1 x1 = new c1();
      c1 y1 = new c1(false);
      void n() { }
      static boolean b = true;
}
```

) computes '`c true c true c false c true c false c true c false true a c`'.

9.7.6 Recursion of static values and constructors

A class can have static value members of its own type. Similarly a constructor can have parameters belonging to the class for which it is a constructor. Quite complex naming mechanisms can result in this case. In JCF54= JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
static c1 x1 = new c1();
static c1 x2 = x1;
static c1 x3 = x2;
static c1 x4 = x3;
static c1 x5 = new c1(x4);
static c1 x6 = x5._u;
static boolean b = x6 instanceof c1;
public static void m(){
      bcop.p(x1.equals(x4));
      bcop.p(c1.x1.empty);
      bcop.p(c1.x2.empty);
      bcop.p(x5.empty);
      bcop.p(x5._u.empty);
      bcop.p(b);
      bcop.p(x6._u.empty);
      }
}

class c1 {
boolean empty;
c1 _u;
static c1 x1 = new c1();
static c1 x2 = new c1(x1.empty);
c1(){empty = true;}
c1(boolean b){empty = false;}
c1(c1 u){empty = false;_u = u;}
}
```

) the use of constructor methods is illustrated. JCF54 computes the following result:

```
true
true
false
false
true
true
```

```
java.lang.NullPointerException
        at c.m(c.java:18)
        at s.main(s.java:2)
```

9.7.7 A complex example

The next example summarizes a multitude of options for constructing objects and using their names. JCF55 = JCFpd ∪ `file:c.java` (

```
import tjvap.*;

public class c {static void m(){c1.m1();}}

class c1 {
static void m1(){
      X x = new X();x._b = false;bcop.p(x._b);
      bcop.p(x._c);x._c = true;
      bcop.p(X._f);
      X z = new X(true,false);bcop.p(! z._c);
      X u = new X(z);bcop.p(! u._b);
      x =  z.next(u); bcop.p(x._c);
      bcop.p(x._b);
      bcop.p(X._f);
      }
}

class X {
public boolean _b,_c = true;
private boolean _d = false;
static private boolean _e = false;
static public boolean _f = true;
X(){_e = !_e; _d = _e;}
X(boolean b,boolean c){_b = b;_c = c;_d = b && c;}
X(X x){_b = x._c;_c = x._b;_d = _e;}
X next(X x){X y = new X(true,true);y = x;
      y._b = this._c  = y._d = y._e = y._f = false;
      return y;}
}
```

). JCF55 computes outputs:

```
false
true
true
true
true
true
```

```
false
false
```

9.8 Booleans as objects

Java allows one to view values of basic data types as objects. This requires built-in operators. The (name of) the corresponding class need not be introduced. Such classes are final and cannot be extended.

In the example below the case of booleans is reviewed, the integers having been added just to illustrate the minuscule difference.

JCF56 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {static void m(){
          c1.m1();c2.m1();}
        }

class c1 {static Boolean b = new Boolean(true);
          static Boolean c = new Boolean(false);
          static boolean d = b.booleanValue();
          static boolean e = c.booleanValue();
          static void m1(){
                bcop.pw(b instanceof Object);
                bcop.pw(b instanceof Boolean);
                bcop.pw(d);bcop.pw(e);
                bcop.pw(new Boolean(!e).booleanValue());cco.aw();
                bcop.pw(new Boolean(!
                          new Boolean(!
                           new Boolean(!
                            new Boolean(true)
                             .booleanValue())
                           .booleanValue())
                          .booleanValue())
                         .booleanValue());
                        cco.b();
          }
}
class c2 {static Integer b = new Integer(0);
          static Integer c = new Integer(1);
          static int d = b.intValue();
          static int e = c.intValue();
          static void m1(){
                bcop.pw(b instanceof Object);
                bcop.pw(b instanceof Integer);
                bcop.pw(d == 0);bcop.pw(e == 0);
```

```
                    bcop.pw(1 == new Integer(1+e).intValue());cco.b();
              }
      }
```

) The result of execution of JCF56 reads

```
true true true false true a false b
true true true false false b
```

9.9 Interfaces

Interfaces generalize the concept of a class without class fields and class methods. An interface can be thought of as the union of the classes having been declared to implement the interface.

9.9.1 Objects in interfaces

The next example indicates how an interface can be declared and how objects in different classes can be instances of the same interface.

JCF57 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
      static void m(){
            c1 x = new c1();
            c2 y = new c2();
            x.m1();x.m2();x.m3();x.m4();cco.d();
            y.m1();y.m2();y.m5();y.m6();cco.d();

            i p;
            p = x;
            i q = y;
            p.m1();bcop.pw(p.m2());cco.e();
            q.m1();bcop.pw(q.m2());cco.e();

            i r = p;p = q;q = r;
            p.m1();bcop.pw(p.m2());cco.f();
            q.m1();bcop.pw(q.m2());cco.f();
            }
      }

class c1 implements i {
      public void m1(){bcop.pw(true);}
      public boolean m2(){bcop.pw(true);return true;}
      public void m3(){cco.aw();}
```

```
      public boolean m4(){cco.aw();return true;}
      }

class c2 implements i {
      public void m1(){bcop.pw(false);}
      public boolean m2(){bcop.pw(false);return false;}
      public void m5(){cco.bw();}
      public boolean m6(){cco.bw();return false;}
      }

interface i {void m1();boolean m2();}
```

) The following outputs are computed:

```
true true a a d
false false b b d
true true true e
false false false e
false false false f
true true true f
```

9.9.2 Interfaces as method parameter types

An extended example indicates that interfaces may serve as parameter types just as classes may do. A non-trivial example of 'instanceof' is possible when interfaces are used. JCF58 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
      static c1 x = new c1();
      static c2 y = new c2();
      static void g(i p,j q){
            p.m1();p.m2();q.m3();q.m4();}
      static void m(){
            x.m1();x.m2();x.m3();x.m4();cco.dw();
            y.m1();y.m2();y.m3();y.m4();cco.dw();

            i p;j q;
            p = x;q = x;
            g(p,q);bco.t();
            p = y;q = y;
            g(p,q);bco.f();

            i p1;j q1;
            bcop.pw(p instanceof c1);bcop.p(p instanceof c2);
            bcop.pw(p.equals(x));bcop.p(p.equals(y));
```

```
            bcop.pw(p.equals(q));bcop.pw(x.equals(y));cco.d();
            p1 = x;q1 = x;
            bcop.pw(p1 instanceof c1);bcop.p(p1 instanceof c2);
            bcop.pw(p1.equals(x));bcop.p(p1.equals(y));
            bcop.pw(p1.equals(q));bcop.pw(p1.equals(q1));cco.d();
            p1 = p;q1 = q;
            g(p1,q1);bco.t();
            }
      }

class c1 implements i,j {
      public void m1(){bco.t();}
      public void m2(){bco.f();}
      public void m3(){m5();}
      public void m4(){cco.aw();}
      private void m5(){bco.t();cco.aw();}
      }

class c2 implements i,j {
      private void m0(){}
      public void m1(){cco.aw();}
      public void m2(){cco.bw();}
      public void m3(){cco.bw();cco.aw();}
      public void m4(){cco.cw();}
      }

interface i {void m1();void m2();}
interface j {void m3();void m4();}
```

) The following outputs are computed:

```
true
false
true
a a d a b b a c d true
false
true
a a true
a b b a c false
false true
false true
true false d
true false
true false
false true d
```

```
a b b a c true
```

9.9.3 Interfaces with instance fields

The role of instance fields in interfaces is an interesting one. In the process of assigning to an interface name (i.e. a variable defined for an interface type) an object, the fields of that object (at least the fields mentioned in the interface) will be updated.

Constant value initialization

Like a class, an interface will be initialized when necessary. Subsequent use of interface names will override the fields of an object for a class implementing the interface with the constants occurring in the interface description.

JCF59 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
      static void m(){
            c1 x1 = new c1();
            c2 x2 = new c2();
            bcop.pw(x1.b);bcop.pw(x1.c);
            bcop.pw(x2.b);bcop.pw(x2.c);cco.a();
            i p,q;p = x1;q = x2;
            bcop.pw(x1.b);bcop.pw(x1.c);
            bcop.pw(x2.b);bcop.pw(x2.c);cco.b();
            bcop.pw(p.b);bcop.pw(p.c);
            bcop.pw(q.b);bcop.pw(q.c);cco.c();
            }
      }

class c1 implements i {
      public boolean b = true;
      public boolean c = true;
      }

class c2 implements i {
      public boolean b = false;
      public boolean c = false;
      }

interface i {
      boolean b = true;
      boolean c = false;
      }
```

) The following outputs are generated:

```
true true false false a
true true false false b
true false true false c
```

Indirect initialization

Initialization of interface fields is similar to static field initialization for classes. JCF60 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
      static void m(){
            c1 x1 = new c1();
            c2 x2 = new c2();
            bcop.pw(x1.b);bcop.pw(x2.b);cco.a();
            i p;p = x1;i q;q = x2;
            bcop.pw(p.equals(x1));bcop.pw(q.equals(x2));
            cco.b();
            bcop.pw(x1.b);bcop.pw(x2.b);cco.c();
            bcop.pw(p.b);bcop.pw(q.b);cco.d();
            c1 y1 = new c1();
            c2 y2 = new c2();
            p = y1;q = y2;
            cco.e();
            }
      }

class c1 implements i {public boolean b = true;}
class c2 implements i {public boolean b = false;}

class init {static boolean in() {cco.fw();return true;}}

interface i {boolean b = init.in();}
```

) The following outputs are generated:

```
true false a
true true b
true false c
f true true d
e
```

The conclusion can be drawn that fields of an interface are initialized the very first time that an object is assigned to any reference for the interface. The results are nevertheless remarkable: the instruction `bcop.pw(p.equals(x1))`

produces `true`, the subsequent instructions `bcop.pw(x1.b)` and `bcop.pw(p.b)` have quite different effects, however, the difference being that the latter action forces `p` to initialize its boolean field (at last).

9.9.4 Interfaces as instance fields

Interfaces now being available,it is possible to produce a minimal pair of classes which, when combined with indentical contexts, provides different outputs. Let

$$\text{JCFxi} = \texttt{class X \{i d;\}}$$

$$\text{JCFxi} = \texttt{class X \{j d;\}}$$

Consider the following JCF, serving as a context to be completed with either JCFxi or JCFxj. JCF61 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {static void m(){
      X x = new X();
      U u = new U();
      x.d = u;
      bcop.p(x.d.b);}
      }

class U implements i,j {boolean b;}

interface i {boolean b = true;}
interface j {boolean b = false;}
```

). The interesting observation is that JCF61 ∪ JCFxi computes `true` whereas JCF61 ∪ JCFxj computes `false`. It seems obvious that a more compact notation of a pair of two classes leading to different outputs in identical contexts is hard to imagine. Classes must have names and named members. If both classes have an empty interface they are equal and cannot generate different outputs in equal contexts. A class with a named member must contain a pair of a member type and a name. The only possible simpler solution to the problem is to have one of the classes memberless. This seems implausible (but we have no proof).

Chapter 10

Class extension: inheritance

10.1 Introduction

Extending a class adds a class which inherits some aspects from the extended class. In the absence of objects class extension inherits static fields and methods from extended classes (super classes) to extending classes (subclasses). If a static field or method is redefined in an extending class the previous definition is said to be overridden. In the chapter on casting it will turn out that overridden fields are not simply 'lost'.

A very elementary example illustrates the phenomenon of class extension and (methods) inheritance. Consider JCF62 = JCFpd ∪ `file:c.java` (

```
import tjvap.*;

class c extends c1 {
static void m() {bco.t();}
}

class c1 extends c2 { }
class c2 extends c3 { }
class c3 { }
```

) which produces `true`.[1] An interesting form of inheritance is observed in a minor modification of the previous example: JCF62 = JCFpd ∪ `file:c.java` (

```
import tjvap.*;

class c extends c1 {
}
```

[1] It is impossible to change this example by having class `c3` an extension of class `c1` as well. The compiler will reject the JCF because of the presence of cyclic class inheritance.

```
class c1 extends c2 {
static void m() {bco.f();}
}

class c2 extends c3 {
}

class c3 { }
```

) which produces `false`. Inheritance is present in the ability of class `c` to accept a method call `c.m()` which it inherits from class `c1`.

10.2 Inheritance and overriding for static fields

Using static boolean variables the following form of inheritance is possible JCF63 = JCFpd ∪ `file:c.java` (

```
import tjvap.*;

class c extends c1 { }

class c1 extends c2 {static void m() {
      bcop.pw(a);bcop.pw(b);bcop.pw(c);cco.a();
      bcop.pw(d);bcop.pw(e);bcop.pw(!f);cco.b();
      }
}

class c2 extends c3 {
      static boolean b = true;
      static boolean c = false;
      static boolean d = !c3.d;
}

class c3 {static boolean
      a = false, b = false, c = false,
      d = true, e = true, f = true;
}
```

) which produces

```
false true false a
false true false b
```

10.3 Class extension and static variable initialization

Class extension has some impact on static variable initialization. Consider JCF64 = JCFpd ∪ `file:c.java` (

```
import tjvap.*;

class c extends c1 {
      static boolean c() {cco.cw();return false;}
      static boolean cc = c();
}

class c1 extends c2 {static void m() {
      bcop.pw(c.cc);bcop.pw(b);bcop.pw(ccc);bcop.p(d);bco.t();
      bcop.pw(c4.d);bcop.pw(c.c());
      bcop.pw(c5.d);bco.f();
      }
}

class c2 extends c3 {
      static boolean b = basis();
      static boolean ccc =  a;
      static boolean d = !b;
}

class basis {static boolean basis() {cco.cw();return true;}}

class c3 extends basis {
      static boolean
      a = a(),b = b();
      private static boolean a() {cco.aw();return basis();}
      private static boolean b() {cco.bw();return !basis();}
}

class c4 extends basis {static boolean d = true;}

class c5 extends basis {static boolean d = d();
      static boolean d() {cco.dw();return false;}
}
```

) which produces

```
a c b c c c false true true false
true
true c false d false false
```

10.4 Class extension and static method overriding

Static methods, also termed class methods, can be overridden just like static fields. The following JCF takes several aspects into account. JCF65 = JCFpd ∪ `file:c.java` (

```
import tjvap.*;

class c {
      static void m() {
            bcop.pw(c1.m());bcop.pw(c2.m());bcop.p(c3.m());}
}

class c1 extends c2 {
      static boolean b = true;
}

class c2 extends c3 {
      static boolean b = false;
      static boolean m() {cco.aw();return b;}
}

class c3 {
      static boolean m() {cco.cw();return b;}
      static boolean b = true;
}
```

) which produces

```
a false a false c true
```

Chapter 11

Combining objects and inheritance

The combination of objects (class instances) and inheritance (class extension) contains the most interesting aspects of object orientation. In particular the super-class constructor call, 'the instanceof' test, overriding instance methods and fields require class extension as well as class instantiation.

11.1 Overloading and class extension

Class extension interacts with overloading because overloaded methods may differ by having arguments from different but comparable classes. JCF66 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
static void f(c1 x) {cco.aw();}
static void f(c2 x) {cco.bw();}
static void f(c3 x) {cco.cw();}
static void m(){
     c1 x1 = new c1();
     c2 x2 = new c2();
     c3 x3 = new c3();
     f(x1);f(x2);f(x3);bco.t();
     c2 y2;
     c3 y3,z3;
           y2 = x1;y3 = x1;
           z3 = x2;
     f(y2);f(y3);bco.t();
     f(z3);bco.f();
```

```
     }
}

class c1 extends c2 { }
class c2 extends c3 { }
class c3 { }
```

) which produces

```
a b c true
b c true
c false
```

The next example also produces the same output it would without the class extensions. JCF67 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
static c2 g(c1 x) {cco.aw();return new c2();}
static c3 g(c2 x) {cco.bw();return new c3();}
static c1 g(c3 x) {cco.cw();return new c1();}
static void m(){
     c1 x1 = new c1();
     c2 x2 = new c2();
     c3 x3 = new c3();
     g(x1);g(x2);g(x3);bco.t();
     g(g(x1));g(g(x2));g(g(x3));bco.f();
     g(g(g(x1)));g(g(g(x2)));g(g(g(x3)));bco.t();
     }
}

class c1 extends c2 { }
class c2 extends c3 { }
class c3 { }
```

) which produces

```
a b c true
a b b c c a false
a b c b c a c a b true
```

11.2 Inheritance of instance fields

As a point of departure consider JCF68 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;
```

```
public class c {
static c1 x = new c1();
public static void m(){x.m();}
}

class c1 {
static boolean q = true;
public void m(){bcop.p(q);}
}
```

), this JCF executes printing `true`. The print instruction is now moved to a class extension. JCF69 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
static c2 x = new c2();
public static void m(){x.m();}
}

class c1 {
static boolean q = true;
}

class c2 extends c1 {
public void m(){bcop.p(c1.q);}
}
```

). This JCF has the same output as the previous one. The following modification is legal as well, leading to output `true` once more: JCF70 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
static c2 x = new c2();
public static void m(){x.m();}
}

class c1 {
static boolean q = true;
}

class c2 extends c1 {
public void m(){bcop.p(q);}
}
```

). Explicit reference to the parameter of the print instruction can be omitted because the extending class inherits the parameter from the extended class.

The boolean can also be introduced as a non-static variable. It will then feature as a non-static variable for the extending class too. JCF71 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
static c2 x = new c2();
public static void m(){x.m();}
}

class c1 {
boolean q = true;
}

class c2 extends c1 {
public void m(){bcop.p(q);}
}
```

). The boolean may be taken from class `c1` explicitly, even though an object of that class has not been constructed explicitly. The output remains the same all the time. JCF72 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
static c2 x = new c2();
public static void m(){x.m();}
}

class c1 {
boolean q = true;
}

class c2 extends c1 {
public void m(){bcop.p(c1.q);}
}
```

)

11.3 Overriding instance fields

A more involved version of the previous examples is: JCF73 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;
```

```
public class c {
static c2 x = new c2();
public static void m(){x.m();}
}

class c1 {
boolean q = true;
}

class c11 extends c1 {}
class c12 extends c11 {}
class c13 extends c12 {}
class c14 extends c13 {}

class c2 extends c14 {
public void m(){bcop.p(q);}
}
```

). The effects of overriding can be illustrated thus: JCF74 = TCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
static c2 x = new c2();
public static void m(){x.m();}
}

class c1 {
boolean q = true;
}

class c11 extends c1 {}

class c12 extends c11 {
boolean q = false;}

class c13 extends c12 {
boolean q = true;}

class c14 extends c13 {
boolean q = false;}

class c2 extends c14 {
public void m(){
      bcop.p(q);bcop.p(c1.q);
      bcop.p(c11.q);bcop.p(c12.q);}
}
```

). The execution of TCF74 produces

```
false
true
true
false
```

11.3.1 Changing inherited static fields

Dynamic aspects of overriding are worth a test as well: TCF75 = TCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
static c2 x = new c2();
public static void m(){x.m();}
}

class c1 {
static boolean q = true;
}

class c2 extends c1 {
public void m(){
            bcop.pw(q);bcop.pw(c1.q);bcop.p(c2.q);
      c1.q = false;
            bcop.pw(q);bcop.pw(c1.q);bcop.p(c2.q);
      q = true;
            bcop.pw(q);bcop.pw(c1.q);bcop.p(c2.q);
      c2.q = false;
            bcop.pw(q);bcop.pw(c1.q);bcop.p(c2.q);
      }
}
```

) produces

```
true true true
false false false
true true true
false false false
```

11.3.2 Changing inherited instance fields

Exactly the same output results from the following example in which the field `q` is non-static. TCF76 = TCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
static c2 x = new c2();
public static void m(){x.m();}
}

class c1 {
boolean q = true;
}

class c2 extends c1 {
public void m(){
            bcop.pw(q);bcop.pw(c1.q);bcop.p(c2.q);
      q = false;
            bcop.pw(q);bcop.pw(c1.q);bcop.p(c2.q);
      this.q = true;
            bcop.pw(q);bcop.pw(c1.q);bcop.p(c2.q);
      ((c1) this).q = false;
            bcop.pw(q);bcop.pw(c1.q);bcop.p(c2.q);
      }
}
```

).

The next example illustrates the incremental introduction of non-static values along a class extension hierarchy. TCF77 = TCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
static c2 x = new c2();
public static void m(){x.m();}
}

class c1 {
boolean q = true;
}

class c11 extends c1 {
boolean q1 = !q;}

class c12 extends c11 {
boolean q2 = !q1;}

class c13 extends c12 {
boolean q3 = true;}
```

```
class c14 extends c13 {
boolean q4 = false;}

class c2 extends c14 {
public void m(){bcop.pw(q);bcop.pw(q1);
               bcop.pw(q2);bcop.pw(q3);bcop.p(q4);}
}
```

) produces

```
true false true true false
```

11.3.3 Overriding resolution

The following example illustrates that static fields and instance fields can override one another. TCF78 = TCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
      static void m(){
            c3 x3 = new c3();
            x3.m();}
}

class c1 {
      public static boolean b = true;
      public boolean c = true;}

class c2 extends c1 {
      private boolean b = false;
      static boolean c = false;
}

class c3 extends c2 {
      boolean c = false;
      void m() {c3 y3 = new c3();bcop.pw(c1.b);bcop.pw(c);
                bcop.pw(c2.c);bcop.pw(((c1) y3).c);
                bcop.p(super.c);}
}
```

). The output of this TCF reads

```
true false false true false
```

11.3.4 Private instance fields

Overriding can be done with a private access specifier thus removing the accessibility of a variable that has been introduced in a superclass. This is illustrated in TCF79 = TCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
static c2 x = new c2();
public static void m(){x.m();}
}

class c1 {
boolean q = true;
}

class c11 extends c1 {
private boolean q = false;
boolean r = q;
boolean s = ((c1) this).q;
}

class c12 extends c11 {
boolean q = true;
}

class c2 extends c12 {
public void m(){bcop.pw(q);bcop.pw(r);bcop.p(s);}
}
```

), which produces

```
true false true
```

The following example combines Java's defaults for static fields with overriding using a private field. TCF80 = TCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
static c3 x = new c3();
public static void m(){x.m();}
}

class c1 {
private boolean q = true;
}
```

```
class c2 extends c1 {
boolean q;
}

class c3 extends c2 {
public void m(){bcop.p(q);}
}
```

) produces `false`. Changing private into public one obtains: TCF81 = TCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
static c3 x = new c3();
public static void m(){x.m();}
}

class c1 {
public boolean q = true;
}

class c2 extends c1 {
boolean q;
}

class c3 extends c2 {
public void m(){bcop.p(q);}
}
```

) which computes `false` as well! This is a significant illustration of the publication assumption.

11.4 Inheritance and constructors

The combinaton of inheritance and constructors is of great importance. The first example is TCF82 = TCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
static c2 x = new c2();
public static void m(){x.m();}
}

class c1 {
```

```
static boolean q = true;
c1() {q = false;bco.t();}
}

class c2 extends c1 {
public void m(){bcop.p(c1.q);}
}
```

) produces `true` followed by `false`. The effect of adding parameters is drastic: TCF83 = TCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
static c2 x = new c2();
public static void m(){x.m();}
}

class c1 {
static boolean q = true;
c1(boolean x) {q = false;bco.t();}
}

class c2 extends c1 {
public void m(){bcop.p(c1.q);}
}
```

) is rejected by the compiler with the message:

```
c.java:13: No constructor matching c1() found in class c1.
      class c2 extends c1 {
```

Interestingly this can be remedied by adding a constructor for class `c2`: TCF84 = TCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
static c2 x = new c2();
public static void m(){x.m();}
}

class c1 {
static boolean q = true;
c1(boolean x) {q = false;bco.t();}
}

class c2 extends c1 {
```

```
c2() {super(true);}
public void m(){bcop.p(c1.q);}
}
```

). JCF85 produces `true` followed by `false` again.

11.4.1 Using the default super-class constructor call

Consider the following modification of JCF84. JCF85 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
static c2 x = new c2();
public static void m(){
      cco.bw();x.m();bco.f();
      c2 y = new c2(c2.q);cco.e();}
}

class c1 {
static boolean q = true;
c1() {q = false;bco.t();}
c1(boolean b) {cco.cw();q = false;bcop.pw(!b);}
}

class c2 extends c1 {
c2() {cco.aw();}
c2(boolean b) {cco.dw();q = !q;}
public void m(){bcop.pw(c1.q);}
}
```

). This JCF produces

```
true
a b false false
true
d e
```

11.4.2 Avoiding the default super-class constructor call

In order to use a super class constructor with arguments one can explicitly call it as the first action of a constructor body. It is called with the name 'super' and the intended parameter list. JCF86 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
static c2 x = new c2();
```

```
public static void m(){
      cco.bw();x.m();bco.f();
      c2 y = new c2(c2.q);cco.e();}
}

class c1 {
static boolean q = true;
c1() {q = false;bco.t();}
c1(boolean b) {cco.cw();q = false;bcop.pw(!b);}
}

class c2 extends c1 {
c2() {super(false);cco.aw();}
c2(boolean b) {super(!b);cco.dw();q = !q;}
public void m(){bcop.p(c1.q);}
}
```

). This JCF produces

```
c true a b false
false
c false d e
```

11.5 Super and inheritance

The keyword 'super' can be used to indicate the parent object of 'this'. There is no 'grand super' to retrieve the 'super of super'. Examples of its use are in JCF87 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class T extends TV {
      boolean b = true;
      void g() {cco.aw();bcop.pw(b);}
      void h() {cco.aw();}
}

class F extends T {
      boolean b = false;
      void g() {cco.bw();bcop.pw(b);bcop.pw(super.b);}
}

class M extends F {
      boolean b = true;
      void g() {cco.cw();}
      void h() {super.h();cco.bw();bcop.pw(b);super.g();}
```

```
}

class D extends M {
      boolean b = false;
      void g() {cco.dw();}
      void h() {cco.dw();super.h();}
}

class C extends D {
      boolean b = true;
      void g() {cco.ew();h();super.h();super.g();}
      void f() {C x = this;D y = this;
           bcop.pw(b);x.g();bco.t();
           bcop.pw(!super.b);y.g();}
}

class c {static void m(){
      T T = new T();
      F F = new F();
      M M = new M();
      D D = new D();
      C C = new C();

      T.g();T.h();bco.t();
      F.g();F.h();bco.f();
      M.g();M.h();bco.t();
      D.g();D.h();bco.f();
      C.g();bco.t();
      C.h();bco.f();
      C.f();bco.f();
      }
}

abstract class TV { }
```

). The output of this JCF reads

```
a true a true
b false true a false
c a b true b false true true
d d a b true b false true false
e d a b true b false true d a b true b false true d true
d a b true b false true false
true e d a b true b false true d a b true b false true d true
true e d a b true b false true d a b true b false true d false
```

11.6 Overriding public and private methods

In this section the publication consistency assumption PCA from section 4.3.2 will be refuted. The counterexample is a variation on an example given by Bart Jacobs ([8]). It turns out to be the case that private methods are protected against being overridden, a protection not enjoyed by private methods. Consider JCF88 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
static void m() {
      B b = new B();
      A a = b;
      a.f();
      }
}

class A {
void f() {g();}
private void g() {
      cco.c();}
}

class B extends A {
public void g() {
      cco.d();}
}
```

). This JCF wiil produce output `d` after compilation and execution. The modification JCF89 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
static void m() {
      B b = new B();
      A a = b;
      a.f();
      }
}

class A {
void f() {g();}
private void g() {
      cco.c();}
}
```

```
class B extends A {
public void g() {
      cco.d();}
}
```

), however, leads to output `c`. The difference between JCF88 and JCF89 is in a keyword `public` that changes to `private` only. Therefore the example refutes PCA.[1]

11.6.1 Class extension hierarchies

The examples of this section illustrate some of the most stereotypical class definition and extension formats.[2]

Standard class introduction

The next example provides a completely standard class definition involving a parametrized constructor (`X(-,-,-,-)`); JCF90 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class X {
boolean _b1,_b2,_b3,_b4;
X(boolean b1,boolean b2,boolean b3,boolean b4) {
        _b1 = b1;_b2 = b2;_b3 = b3;_b4 = b4;}
boolean sign() {boolean b = true;
        if(_b1){b = !b;};if(_b2){b = !b;}
        if(_b3){b = !b;};if(_b4){b = !b;}
        return b;}
void flip() {_b1 = !_b1;_b2 = !_b2;_b3 = !_b3;_b4 = !_b4;}
}

class c {
static void m(){
      boolean a,b,c = true;
      a = true || b;
      X A,B,C;
      A = new X(true,false,true,false);
      b = !A._b3;
            bcop.p(b);bcop.p(A.sign());
```

[1] If we must choose one of the two outputs to be 'unnatural', in accordance with the earlier statement that PCA may be considered a desirable property, we consider the processing of JCF88 to be less natural. It is not clear why calling a method within class `A` is influenced by this method having been overridden elsewhere (i.e. in class `B`).

[2] Applets and input structures constitute classes which are usually not extended in this fashion.

```
        A.flip();B = A;B._b3 = !B._b2;
              bcop.p(A._b3);bcop.p(A.sign());
        C = new X(a,b,c,c);
              cco.a();bcop.p(C._b1);
        B = C;B._b1 = !B._b1;
              bcop.p(C._b1);bcop.p(C.sign());
        }
    }
```

). Its output is

```
    false
    true
    false
    true
    a
    true
    false
    true
```

Standard class extension

An extension of this class adds two additional boolean parameters. The method `sign()` must be overridden, as well as the method `flip`. JCF91 = JCFpd ∪ `file:c.java`(

```
    import tjvap.*;

    class X {
    boolean _b1,_b2,_b3,_b4;
    X() {_b1 = _b2 = _b3 = _b4 = false;}
    X(boolean b1,boolean b2,boolean b3,boolean b4) {
            _b1 = b1;_b2 = b2;_b3 = b3;_b4 = b4;}
    boolean sign() {boolean b = true;
            if(_b1){b = !b;};if(_b2){b = !b;}
            if(_b3){b = !b;};if(_b4){b = !b;}
            return b;}
    void flip() {_b1 = !_b1;_b2 = !_b2;_b3 = !_b3;_b4 = !_b4;}
    }

    class Y extends X {
    boolean _b5,_b6;
    Y(){super();_b1 = _b2 = _b3 = _b4 = _b5 = _b6 = true;}
    Y(boolean b) {super(true,true,b,!b);_b5 = false;_b6 = false;}
    Y(boolean b1,boolean b2,boolean b3,boolean b4,
      boolean b5,boolean b6){super(b1,b2,b3,b4);
            _b5 = b5;_b6 = b6;}
```

```
boolean sign() {boolean b = true;
                if(_b1){b = !b;}if(_b2){b = !b;}
                if(_b3){b = !b;}if(_b4){b = !b;}
                if(_b5){b = !b;}if(_b6){b = !b;}
                return b;}
static boolean sign(Y y) {boolean b = true;
                if(y._b1){b = !b;}if(y._b2){b = !b;}
                if(y._b3){b = !b;}if(y._b4){b = !b;}
                if(y._b5){b = !b;}if(y._b6){b = !b;}
                return b;}
void flip() {_b1 = !_b1;_b2 = !_b2;_b3 = !_b3;_b4 = !_b4;
             _b5 = !_b5;_b6 = !_b6;}
Y yflip() {_b1 = !_b1;_b2 = !_b2;_b3 = !_b3;_b4 = !_b4;
              _b5 = !_b5;_b6 = !_b6;return this;}
Y copy() {Y y = new
          Y(this._b1,this._b2,this._b3,this._b4,this._b5,this._b6);
          return y;}
boolean cosign(){this.flip();return this.sign();}
}

class c {
static void m(){
      Y A,B,C,D,E,F;
      E = new Y(true,true,true,true,true,false);
            bcop.pw(E.sign());bcop.pw(Y.sign(E));
      F = new Y(true,true,true,true,false,false);
            bcop.pw(F.sign());bcop.pw(Y.sign(F));
      E._b1 = false;
            bcop.pw(E.sign());bcop.pw(Y.sign(E));
      A = new Y();
            bcop.pw(A._b2);bcop.pw(!A._b6);
      B = new Y(true);
            bcop.pw(B._b2);bcop.pw(!B._b6);
      C = new Y(true,false,true,A._b3,B._b5,B.yflip().cosign());
            bcop.pw(C._b2);bcop.pw(C._b6);
            cco.b();
      D = B.copy();
      B.yflip().yflip().flip();
            bcop.pw(B._b2);bcop.pw(D._b2);
      A = B.yflip().yflip().yflip();
      boolean b = A._b2;
      C = new Y(true);
            bcop.pw(A._b2);bcop.pw(B._b2);
            bcop.pw(C.sign());bcop.pw(C.cosign());
            bcop.pw(D._b2);cco.e();
      C._b4 = !C._b4;
```

```
            bcop.pw(D.sign());bcop.pw(D.cosign());
            bcop.pw(C.sign());bcop.pw(C.cosign());
            bcop.pw(C.sign());bcop.pw(C.cosign());
      C._b5 = !C._b5;
            bcop.pw(C.sign());bcop.pw(C.cosign());
            cco.a();
      D.flip();
            bcop.pw(D.sign());bcop.pw(Y.sign(D));bcop.pw(D.cosign());
            bcop.pw(D.cosign());bcop.pw(D.sign());bcop.pw(Y.sign(D));
            cco.c();bcop.pw(D._b1);cco.a();
      D._b1 = false;
            bcop.pw(D._b1);
            bcop.pw(Y.sign(D));bcop.pw(D.sign());
            bcop.pw(D.yflip().sign());
      D._b1 = !D._b1;
            bcop.pw(Y.sign(D));bcop.pw(D.sign());
            cco.b();
      }
}
```

). Its output is

```
false false true true true true true false true true false false b
false true true true false false true e
false false true true true true false false a
false false false false false false c
true a
false true true true false false b
```

11.7 Interfaces and class extensions

The connection between interfaces and class extension merits some attention. The fields of an interface should be initialized. Assigning an interface name to an object invokes an update of the corresponding fields in the object. In the example below this update works after an assignment of an object to a superclass reference (the names `r` and `s`) which has already influenced the value of the fields. JCF92 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
      static void m(){
            c1 x = new c1();
            c2 y = new c2();
            C z = new C();
            bcop.pw(x.b);x.m1();x.m2();x.m3();cco.d();
```

```
            bcop.pw(y.b);y.m1();y.m2();y.m4();cco.d();
            bcop.pw(z.b);z.m1();z.m2();cco.d();cco.d();

            i p,q,r,s;
            C u,v,w;
            p = x;q = y;
            u = x;v = y;
            w = x;r = w;
            w = y;s = w;
            bcop.pw(x.b);bcop.pw(x.c);x.m1();bcop.p(x.m2());
            bcop.pw(p.b);bcop.pw(p.c);p.m1();bcop.p(p.m2());
            bcop.pw(u.b);bcop.pw(u.c);u.m1();bcop.p(u.m2());
            bcop.pw(r.b);bcop.pw(r.c);r.m1();bcop.p(r.m2());
            cco.e();

            bcop.pw(y.b);bcop.pw(y.c);y.m1();bcop.p(y.m2());
            bcop.pw(q.b);bcop.pw(q.c);q.m1();bcop.p(q.m2());
            bcop.pw(v.b);bcop.pw(v.c);v.m1();bcop.p(v.m2());
            bcop.pw(s.b);bcop.pw(s.c);s.m1();bcop.p(s.m2());
            cco.f();
            }
      }

class c1 extends C implements i {
      public boolean b = true;
      public boolean c = true;
      public void m1(){bcop.pw(true);}
      public boolean m2(){bcop.pw(true);return true;}
      public void m3(){cco.aw();}
      }

class c2 extends C implements i {
      public boolean b = false;
      public boolean c = false;
      public void m1(){bcop.pw(false);}
      public boolean m2(){bcop.pw(false);return false;}
      public void m4(){cco.bw();}
      }

class C implements i {
      public boolean b = true;
      public boolean c = true;
      public void m1(){bcop.pw(true);bcop.pw(false);}
      public boolean m2(){bcop.pw(true);bcop.pw(false);return true;}
      }
```

```
interface i {
      boolean b = true;
      boolean c = false;
      void m1();
      boolean m2();}
```

) The following outputs are computed:

```
true true true a d
false false false b d
true true false true false d
d
true true true true true
true false true true true
true true true true true
true false true true true
e
false false false false false
true false false false false
true true false false false
true false false false false
f
```

11.7.1 Extending interfaces

Interfaces can extend one another. Numerous examples can be imagined. The following JCF demonstrates the interaction between interface extension and field initialization. The equals-test is investigated later. It has no side-effects here and can be skipped by readers who prefer not to be distracted by it now. JCF93 = JCFpe ∪ `file:c.java`(

```
import tjvap.*;

class c {static void m(){
      U u = new U();
      ico.pw(u.b);ico.p(u.c);
      i p = u;
      bcop.pw(p.equals(u));bcop.p(u.equals(p));
      ico.pw(u.b);ico.p(u.c);
      ico.pw(p.b);ico.p(p.c);
      ico.pw(((U) p).b);ico.p(((U) p).c);
      j q = p;
      ico.pw(u.b);ico.pw(u.c);ico.p(q.c);
      bcop.pw(p.equals(q));bcop.pw(q.equals(p));
      bcop.pw(u.equals(q));bcop.pw(q.equals(u));
      cco.a();
      V v = new V();
```

```
        ico.pw(v.b);ico.p(v.c);
        i pp = v;
        ico.pw(v.b);ico.p(v.c);
        ico.pw(pp.b);ico.p(pp.c);
        j qq = pp;
        }
}

class U implements i,j {int b;int c;}
class V implements i,j {int b = 12;int c =  8;}

interface i extends j {int b = 37;}
interface j {int c = 45;}
```

). The outputs read:

```
0 0
true true
0 0
37 45
0 0
45
true true true true a
12 8
12 8
37 45
12 8
45
```

Part IV

Object identity and object existence

Chapter 12

Casting

Objects can be assigned to variables of different classes. It is essential that the object has been created in the class of the variable or in an extension of it. Otherwise the compiler will complain. The presence of casting in Java has only a technical rationale. Java compilers are simpler to build and cheaper to run if programmers are forced to add explicit casting information in certain cases. The notation for casting is as follows: '`(K) q`' is an expression denoting `q`, temporarily used as if its type were `K`.

Objects can only be cast to types of which they are an instance. For this reason the interaction between casting and the 'instanceof' test is important.

12.1 Implicit casting

If an object is assigned to a variable of a different class this will be called implicit casting. It is implicit because of the absence of an explicit casting operator. For instance: JCF94 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c extends c1 {
      static void m() {
            c1 x = new c1();x.M();
            c2 y1 = new c2();y1.M();
            c2 y2 = x;y2.M();cco.a();
      }
}

class c1 extends c2 {
     void M() {bcop.pw(b);}
}

class c2 extends c3 {
```

```
      void M() {bcop.pw(!b);}
}

class c3 {
      static boolean b = true;
}
```

) Execution of the compiled version of this JCF produces:

```
true false true a
```

It can be concluded that if an object is the target of a method, the type of the name used to point at it is immaterial.

12.2 Explicit casting

12.2.1 Overriding resolution and overloading resolution

The following long example illustrates the effect of casting on the selection of method bodies and fields) selection.[1] JCFubc contains a hierarchy of classes, involving a chain of method overridings and of field overridings. JCFubc = `file:Ubc.java` (

```
import tjvap.*;

class c1 extends c2 {boolean b = true,c = false;
      boolean f() {cco.aw();return true;}}

class c2 extends c3 {boolean b = false,c = false;
      boolean f() {cco.bw();return false;}}

class c3 extends c4 {boolean b = true,c = true;
      boolean f() {cco.cw();return true;}}

class c4 extends c5 {boolean b = false,c = true;
      boolean f() {cco.dw();return false;}}

class c5 extends c6 {boolean b = true,c = false;
      boolean f() {cco.ew();return true;}}

class c6 extends c7 {boolean b = false,c = false;
      boolean f() {cco.fw();return false;}}

class c7 extends U {boolean b = true,c = true;
      boolean f() {cco.gw();return true;}}
```

[1] This selection process can be called: overriding resolution.

```
class U {boolean b = false,c = true;
      boolean f() {cco.gw();return false;}}


class F {
static boolean f(c1 x) {
      bcop.pw(x.b);bcop.pw(x.f());cco.aw();return x.b;}

static boolean f(c2 x) {
      bcop.pw(x.b);bcop.pw(x.f());cco.bw();return x.b;}

static boolean f(c3 x) {
      bcop.pw(x.b);bcop.pw(x.f());cco.cw();return x.b;}

static boolean f(c4 x) {
      bcop.pw(x.b);bcop.pw(x.f());cco.dw();return x.b;}

static boolean f(c5 x) {
      bcop.pw(x.b);bcop.pw(x.f());cco.ew();return x.b;}

static boolean f(c6 x) {
      bcop.pw(x.b);bcop.pw(x.f());cco.fw();return x.b;}

static boolean f(c7 x) {
      bcop.pw(x.b);bcop.pw(x.f());cco.gw();return x.b;}

static boolean f(U x) {
      bcop.pw(x.b);bcop.pw(x.f());cco.hw();return x.b;}
}
```

) An example using JCFubc will be documented, demonstrating different forms of resolution.

12.2.2 Overriding resolution for fields

The following JCF illustrates the way in which Java decides which field to select in the case of a hierachy of classes. JCF95 = JCFpd ∪ JCFubc ∪ `file:c.java`
(

```
import tjvap.*;

class c {
      static void m() {
            c1 y1 = new c1();
            c2 y2 = new c2();
            c3 y3 = new c3();
```

```
            c4 y4 = new c4();
            c5 y5 = new c5();
            c6 y6 = new c6();
            c7 y7 = new c7();
            U y = new U();

      // implicit casting
            bcop.pw(y3.b);bcop.pw(y3.f());        cco.g();
            y4 = y3;bcop.pw(y4.b);bcop.pw(y4.f());cco.g();
            cco.a();

      // explicit casting
            bcop.pw(((c4) y3).b);    cco.gw();
            bcop.pw(((c4) y3).f());  cco.gw();
            bcop.pw(((c5) y3).b);    cco.gw();
            bcop.pw(((c5) y3).f());  cco.g();
            cco.b();

      // repeated explicit casting
            bcop.pw(((c4)(c7) y3).b);      cco.gw();
            bcop.pw(((c4)(c7) y3).f());    cco.gw();
            bcop.pw(((c4)(c7)(c5) y3).b);  cco.g();
            bcop.pw(((c4)(c7)(c5) y3).f());cco.gw();
            bcop.pw(((c5) (U) y3).b);      cco.gw();
            bcop.pw(((c5) (U) y3).f());    cco.g();
            cco.c();

      // overloaded method application - explicit casting
            bcop.pw(F.f(y3));                 cco.g();
            bcop.pw(F.f((c4) y3));            cco.g();
            bcop.pw(F.f((c5) y3));            cco.g();
            bcop.pw(F.f((c4) (c7) (c5) y3));  cco.g();
            cco.d();

      // overloaded method application -implicit casting
            y4 = y3;bcop.pw(F.f(y4));         cco.g();
            y5 = y3;bcop.pw(F.f(y5));         cco.g();
            y6 = y3;bcop.pw(F.f(y6));         cco.g();
            y5 = (c4) y3;bcop.pw(F.f(y5));    cco.g();
            y6 = (c5) y3;bcop.pw(F.f(y6));    cco.g();
            y6 = (c5)(c7) y3;bcop.pw(F.f(y6));cco.g();
            cco.e();
      }
}
```

). This JCF produces

```
true c true g
false c true g
a
false g c true g true g c true g
b
false g c true g false g
c true g true g c true g
c
true c true c true g
false c true d false g
true c true e true g
false c true d false g
d
false c true d false g
true c true e true g
false c true f false g
true c true e true g
false c true f false g
false c true f false g
e
```

12.2.3 Casting and overriding resolution II

The effect of casting on overriding resolution for instance fields is completely different from the effect of casting on method overriding resolution. The folowing example is a second indication. Consider JCF96 = JCFpe ∪ `file:c.java`(

```
import tjvap.*;

class A1 implements j1,i {
      int f = 1;public int g() {return 5;}}
class A2 extends A1 implements j2,i {
      int f = 2;public int g() {return 6;}}
class A3 extends A2 implements j3,i {
      int f = 3;public int g() {return 7;}}
class A4 extends A3 implements j4,i {
      int f = 4;public int g() {return 8;}}

interface j1 {int f = 9;int g();}
interface j2 {int f = 10;int g();}
interface j3 {int f = 11;int g();}
interface j4 {int f = 12;int g();}

interface i {int f = 13;int g();}

class c {static void m(){
```

```
        A1 x1 = new A1();
        A2 x2 = new A2();
        A3 x3 = new A3();
        A4 x4 = new A4();
                                          cco.a();
        ico.pw(x1.f);ico.pw(x1.g());
        ico.pw(x2.f);ico.pw(x2.g());
        ico.pw(x3.f);ico.pw(x3.g());
        ico.pw(x4.f);ico.pw(x4.g());
                                                cco.b();
        ico.pw(((A1)x1).f);ico.pw(((A1)x1).g());
        ico.pw(((A1)x2).f);ico.pw(((A1)x2).g());
        ico.pw(((A1)x3).f);ico.pw(((A1)x3).g());
        ico.pw(((A1)x4).f);ico.pw(((A1)x4).g());
                                                cco.c();
        ico.pw(((A2)x2).f);ico.pw(((A2)x2).g());
        ico.pw(((A2)x3).f);ico.pw(((A2)x3).g());
        ico.pw(((A2)x4).f);ico.pw(((A2)x4).g());
                                                cco.d();
        ico.pw(((A3)x3).f);ico.pw(((A3)x3).g());
        ico.pw(((A3)x4).f);ico.pw(((A3)x4).g());
                                                cco.e();
        ico.pw(((A4)x4).f);ico.pw(((A4)x4).g());
                                                cco.e();
        A1 y1 = (A1) new A2();
        A1 z1 = (A1) new A3();A2 z2 = (A2) new A3();
        A1 u1 = (A1) new A4();A2 u2 = (A2) new A4();A3 u3 = new A4();
        ico.pw(y1.f);ico.pw(y1.g());
        ico.pw(z1.f);ico.pw(z1.g());
        ico.pw(u1.f);ico.pw(u1.g());
        ico.pw(z2.f);ico.pw(z2.g());
        ico.pw(z2.f);ico.pw(z2.g());
        ico.pw(u3.f);ico.pw(u3.g());
                                       cco.f();
        }
}
```

). The output of this JCF reads

```
a
1 5 2 6 3 7 4 8 b
1 5 1 6 1 7 1 8 c
2 6 2 7 2 8 d
3 7 3 8 e
4 8 e
1 6 1 7 1 8 2 7 2 7 3 8 f
```

12.2.4 Casting and overriding resolution III

The effect of casting on overriding resolution for instance fields is different from the effect of casting on non-void method overriding resolution. The effect of casting on overloading resolution is nil. Consider JCF97 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

abstract class TV { }

class T extends TV {
      boolean b = true;
      boolean c() {return true;}
      void g() {cco.aw();}
}

class F extends T {
      boolean b = false;
      boolean c() {return false;}
}

class M extends F {
      boolean b = true;
      boolean c() {return true;}
}

class TFMc {
      static boolean f(T x) {bcop.pw(x.b);cco.aw();return x.c();}
      static boolean f(F x) {bcop.pw(x.b);cco.bw();return x.c();}
      static boolean f(M x) {bcop.pw(x.b);cco.cw();return x.c();}
}

class c {static void m(){
      T T = new T();
      F F = new F();
      M M = new M();

      bcop.pw(T.b);bcop.p(T.c());
      bcop.pw(F.b);bcop.pw(F.c());
      bcop.pw(((T) F).b);bcop.p(((T) F).c());
      bcop.pw(M.b);bcop.pw(M.c());
      bcop.pw(((F) M).b);bcop.p(((F) M).c());

      cco.a();
      bcop.pw(TFMc.f(T));bcop.pw(TFMc.f(F));bcop.pw(TFMc.f(M));

      cco.b();
```

```
            bcop.pw(TFMc.f((T) F));bcop.pw(TFMc.f((F) M));bcop.pw(TFMc.f((T) M));

            cco.c();
            T tf = (T) F;
            F fm = (F) M;
            T tm = (T) fm;
            bcop.pw(TFMc.f(tf));bcop.pw(TFMc.f(fm));bcop.pw(TFMc.f(tm));

            cco.d();
            T tf1 = F;
            F fm1 = M;
            T tm1 = M;
            bcop.pw(TFMc.f(tf1));bcop.pw(TFMc.f(fm1));bcop.pw(TFMc.f(tm1));

            cco.e();
            bcop.pw(TFMc.f((F) tf1));bcop.pw(TFMc.f((M) fm1));
            bcop.pw(TFMc.f((F) tm1));bcop.p(TFMc.f((M) tm1));}
      }
```

). The output of this JCF reads

```
true true
false false true false
true true false true
a
true a true false b false true c true b
true a false false b true true a true c
true a false false b true true a true d
true a false false b true true a true e
false b false true c true false b true true c true
```

12.3 Casting and interfaces

The interaction between explicit casting and interfaces allows some simple experiments.

12.3.1 Casting an interface instance

JCF98 = JCFpd ∪ `file:c.java`(

```
      import tjvap.*;

      class c {
            static void m(){
                  c1 x1 = new c1(),y1 = new c1();
                  c2 x2 = new c2(),y2 = new c2();
```

```
            C z = new C();
            i p,q,r,s,t,u;
            p = (C) x1;q = y1;r = (C) x2;
            p.m1();bcop.p(p.m2());
            q.m1();bcop.p(q.m2());
            r.m1();bcop.p(r.m2());
            s = (c1) p;
            s.m1();bcop.p(s.m2());
            ((i) r).m1();bcop.p(((i) r).m2());
            }
      }

class c1 extends C implements i {
      public boolean b = true;
      public void m1(){cco.aw();}
      public boolean m2(){return true;}
      }

class c2 extends C implements i {
      public boolean b = false;
      public void m1(){cco.bw();}
      public boolean m2(){return false;}
      }

class C implements i {
      public boolean b = false;
      public void m1(){cco.cw();}
      public boolean m2(){return true;}
      }

interface i {
      boolean b = true;
      void m1();
      boolean m2();}
```

) The following outputs are computed:

```
a true
a true
b false
a true
b false
```

12.3.2 Casting to an interface, causing a run-time error

The following JCF is accepted by the compiler. Its execution generates a run-time error, however. The variable `x1` for class `c1` can not be matched with the

interface because its class has not been explicitly declared as implementing the interface. JCF99 = `file:c.java`(

```
import tjvap.*;

class c {
static c1 x1 = new c1();
static void m(){
      c2 x2 = new c2();x1.m1();x2.m1();cco.b();
      i ivar = (i) new  c2();cco.b();
      ivar.m1();cco.b();
      ivar = (i) x1;
      ivar.m1();
      }
}

class c1 {
      c1() {cco.aw();}
      public void m1(){cco.ew(); }
}

class c2 extends c1 implements i {
      c2() {super();}
}
```

) ∪ `file:i.java`(

```
interface i {void m1();}
```

). The result of execution is:

```
a a e e b
a b
e b
java.lang.ClassCastException: c1
        at c.m(c.java:9)
        at s.main(s.java:2)
```

12.4 Casting and object identification

Object identity can be maintained via the 'equals()' method. This built-in method for Java can be used to test whether two object expressions refer to the same object. Because 'equals()' can be overloaded this functionality can be undone by the programmer at will.

12.4.1 The features 'instanceof' and 'equals'

This example explores the effect of the operations `-.equals(-)` and `-.instanceof(-)`. 'equals' can be overridden. If it is not overridden it stands for object equality. 'instanceof' determines if an object is the instantiation of a class. If it is, it is an instantiation of all superclasses as well. JCF100 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

public class c {
static c1 x = new c1();
static c1 y = new c1();
static c2a za = new c2a();
static c2a ua = new c2a();
static c3a va = new c3a();
static c3a wa = new c3a();
static c2b zb = new c2b();
static c2b ub = new c2b();
static c3b vb = new c3b();
static c3b wb = new c3b();
public static void m(){
     x.m();
     y.m();
     if(x.equals(y)){bco.t();} else {bco.f();}
       y = x;
     if(x.equals(y)){bco.t();} else {bco.f();}
       y = new c1();
     if(x.equals(y)){bco.t();} else {bco.f();}
     za.m();
     ua.m();
       x = ua;
     x.m();
     va.m();
       y = wa;
     y.m();
     zb.m();
     ub.m();
       x = ub;
     x.m();
     vb.m();
       y = wb;
     y.m();
     if(x instanceof c2a){bco.t();} else {bco.f();}
     if(x instanceof c2b){bco.t();} else {bco.f();}
     if(x instanceof c3b){bco.t();} else {bco.f();}
     if(x instanceof c1){bco.t();} else {bco.f();}
     if(y instanceof c1){bco.t();} else {bco.f();}
```

```
        }
    }

    class c1 {public void m(){cco.a();}}
    class c2a extends c1 {public void m(){cco.b();}}
    class c3a extends c2a {public void m(){cco.c();}}
    class c2b extends c1 {public void m(){cco.d();}}
    class c3b extends c2b {public void m(){cco.e();}}
```

)
The result of execution of JCF100 consists of the successive outputs

```
a
a
false
true
false
b
b
b
c
c
d
d
d
e
e
false
true
false
true
true
```

12.4.2 Casting related to 'instanceof' and 'equals'

This example deals with the effects of casting in relation to the features 'equals' and 'instanceof' JCF101 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

abstract class V { }

class T extends V {void g() {cco.aw();}}
class F extends T {void g() {cco.bw();}}
class M extends F {void g() {cco.cw();}}
class Mm extends F {void g() {cco.cw();}}
class D extends M {void g() {cco.dw();}}
```

```
class C extends D {void g() {cco.ew();}}

class c {static void m(){
      T T = new T();
      F F = new F();
      M M = new M();
      Mm Mm = new Mm();
      D D = new D();
      C C = new C();

      bcop.pw(T instanceof T);
      bcop.pw(T instanceof F);

      bcop.pw(F instanceof T);
      bcop.pw(F instanceof F);
      bcop.pw(F instanceof M);
      bcop.pw((F) Mm instanceof M);
      bcop.pw((T) D instanceof Mm);
      cco.a();

      bcop.pw((F)(T)(F)(T) M instanceof T);
      bcop.pw((F)(T)(F)(T) M instanceof F);
      bcop.pw((F)(T)(F)(T) M instanceof M);
      bcop.pw((F)(T)(F)(T) M instanceof D);
      cco.b();

      bcop.pw(((T) C).equals((T) C));
      bcop.pw(((T) C).equals(C));
      bcop.pw(((T) C).equals((F) C));
      bcop.pw(((T)(M) C).equals((T)(M) C));
      bcop.pw(((T)(M) C).equals((T) C));
      bcop.pw(((M)(F) C).equals((M)(T) C));
      cco.d();

      bcop.pw(((T) C).equals((T) D));
      cco.e();

      T.g();     F.g();     M.g();     D.g();     C.g();bco.t();
            ((T)F).g();((F)M).g();((M)D).g();((D)C).g();bco.f();

      T T1 = (T) M;T1.g();((F) T1).g();
      F F1 = (F) M;F1.g();((F) F1).g();
      M M1 = (M) M;F1.g();((F) M1).g();
      bco.t();
      }
}
```

). The output of this JCF reads

```
true false true true false false false a
true true true false b
true true true true true true d
false e
a b c d e true
b c d e false
c c c c c c true
```

It can be concluded that casting has no effect on overriding resolution for instance methods.

Chapter 13

Experimenting with garbage collection

Garbage collection takes place if the machine memory is freed from objects that cannot be accessed anymore. If garbage collection is not automatic, it has to be done via program instructions. The objective of this section is to design some elementary experiments related to the ability of the implementation to perform automatic garbage collection. The conclusion is that garbage collection is by no means easy to provoke.

The definition of 'garbage' is not at all trivial. The correct intuition is that objects remain in existence until garbage has been collected, irrespective of the existence of names ('in scope') referring to them. Garbage collection is preceded by the execution of a finalize method which may have been been overridden in the class definition of the objects to be collected. The overriding method body may contain an action marking its execution. That mark is no guarantee that garbage collection has removed an object, however. It should be mentioned that garbage collection is notoriously implementation-dependent and that these experiments cannot simply be generalized to any other configuration.

One may say that objects exist until garbage has been collected. The existence of objects in memory is not primarily determined by the presence of a 'live' reference to the object. Garbage collection removes objects that exist but are no longer needed.

13.1 Binary trees

Binary trees are introduced in JCFbintree.
JCFbintree = JCFpd ∪ `file:treeNode.java`(

```
import tjvap.*;

abstract class binTree {
```

```
boolean isLeaf;
boolean value;
}

class treeLeaf extends binTree {
treeLeaf(boolean _value) {
      isLeaf = true;value = _value;}
}

class treeNode extends binTree {
binTree leftSon,rightSon;
treeNode(boolean _value,binTree _leftSon,binTree _rightSon) {
      isLeaf = false;value = _value;
      leftSon = _leftSon;
      rightSon = _rightSon;
      }
}

class soLo {
binTree bintree;
soLo(binTree _bintree){
      bintree = _bintree;}
protected void finalize() {cco.fw();}
}
```

) which is checked and accepted by the command:

```
javac treeLeaf.java
```

13.2 Large-tree constructors

The next JCF adds a class allowing methods for constructing increasingly large trees. Far bigger trees can be constructed by these methods than will be used in the experments. The same experiments can be modified by incresing the size of the trees involved. This is done by increasing the numerical part of the used method names by 1 or by 2 or.. by 12. The resulting family of experiments will suffice for somewhat faster and bigger machines as well. JCFbintreebuilder = JCFbintree ∪ `file:treeNode.java`(

```
class binTreeBuilder {
static binTree b01() {return new treeLeaf(true);}
static binTree b02() {return new treeNode(false,b01(),b01());}
static binTree b03() {return new treeNode(false,b02(),b02());}
static binTree b04() {return new treeNode(false,b03(),b03());}
static binTree b05() {return new treeNode(false,b04(),b04());}
static binTree b06() {return new treeNode(false,b05(),b05());}
```

```
      static binTree b07() {return new treeNode(false,b06(),b06());}
      static binTree b08() {return new treeNode(false,b07(),b07());}
      static binTree b09() {return new treeNode(false,b08(),b08());}
      static binTree b10() {return new treeNode(false,b09(),b09());}
      static binTree b11() {return new treeNode(false,b10(),b10());}
      static binTree b12() {return new treeNode(false,b11(),b11());}
      static binTree b13() {return new treeNode(false,b12(),b12());}
      static binTree b14() {return new treeNode(false,b13(),b13());}
      static binTree b15() {return new treeNode(false,b14(),b14());}
      static binTree b16() {return new treeNode(false,b15(),b15());}
      static binTree b17() {return new treeNode(false,b16(),b16());}
      static binTree b18() {return new treeNode(false,b17(),b17());}
      static binTree b19() {return new treeNode(false,b18(),b18());}
      static binTree b20() {return new treeNode(false,b19(),b19());}
      static binTree b21() {return new treeNode(false,b20(),b20());}
      static binTree b22() {return new treeNode(false,b21(),b21());}
      static binTree b23() {return new treeNode(false,b22(),b22());}
      static binTree b24() {return new treeNode(false,b23(),b23());}
      static binTree b25() {return new treeNode(false,b24(),b24());}
      static binTree b26() {return new treeNode(false,b25(),b25());}
      static binTree b27() {return new treeNode(false,b26(),b26());}
      static binTree b28() {return new treeNode(false,b27(),b27());}
      static binTree b29() {return new treeNode(false,b28(),b28());}
      static binTree b30() {return new treeNode(false,b29(),b29());}
      }
```

)

13.3 No signs of garbage collection?

The following sequence of examples produces cases in which garbage collection for can easily be imagined, but seems not to take place.

According to most books Java provides no definite claims about when garbage is to be collected.

13.3.1 Big objects and a lack of memory

The next step is to add a class introducing a number of objects, allowing a test about the ability to construct large objects. JCF102 = JCFbintreebuilder ∪ `file:c.java`(

```
      import tjvap.*;

      class c {
      static void m() {
            binTree x14 = binTreeBuilder.b14();cco.aw();
```

```
        soLo y14 = new soLo(binTreeBuilder.b14());cco.a();

        binTree x17 = binTreeBuilder.b17();cco.bw();
        soLo y17 = new soLo(binTreeBuilder.b17());cco.b();

        binTree x17b = binTreeBuilder.b17();cco.cw();
        soLo y17b = new soLo(binTreeBuilder.b17());cco.c();

        binTree x18 = binTreeBuilder.b18();cco.dw();
        soLo y18 = new soLo(binTreeBuilder.b18());cco.d();
        }
}
```

) produces output:

```
a a
b b
Exception in thread "main" java.lang.OutOfMemoryError
```

In principle garbage collection is conceivable as a means for removing all objects having been constructed before `y17b`. Absence of garbage collection follows from the error and the absence of output of the `finalize()` method for objects of the class `soLo`.

13.3.2 Introducing scope restrictions

The next experiment is JCF103 = JCFbintreebuilder ∪ `file:c.java`(

```
import tjvap.*;

class c {
static void m() {
      {binTree x14 = binTreeBuilder.b14();cco.aw();
       soLo y14 = new soLo(binTreeBuilder.b14());cco.a();}

      {binTree x17 = binTreeBuilder.b17();cco.bw();
       soLo y17 = new soLo(binTreeBuilder.b17());cco.b();}

      {binTree x17b = binTreeBuilder.b17();cco.cw();
       soLo y17b = new soLo(binTreeBuilder.b17());cco.c();}

      {binTree x18 = binTreeBuilder.b18();cco.dw();
       soLo y18 = new soLo(binTreeBuilder.b18());cco.d();}
      }
}
```

) produces the same output as the previous one. The names now all dropping out of scope, garbage collection is becoming more likely. Nevertheless the effects are minimal.

13.3.3 Re-using names

The example can be changed by re-using names after their scope has exited. This effectively removes references to the objects these names were pointing to (at least in the example below). The computation produces the same output again, however. JCF104 = JCFbintreebuilder ∪ `file:c.java`(

```
import tjvap.*;

class c {
static void m() {
      {binTree x14 = binTreeBuilder.b14();cco.aw();
       soLo y14 = new soLo(binTreeBuilder.b14());cco.a();}

      {binTree x17 = binTreeBuilder.b17();cco.bw();
       soLo y17 = new soLo(binTreeBuilder.b17());cco.b();}

      {binTree x17 = binTreeBuilder.b17();cco.cw();
       soLo y17 = new soLo(binTreeBuilder.b17());cco.c();}

      {binTree x18 = binTreeBuilder.b18();cco.dw();
       soLo y18 = new soLo(binTreeBuilder.b18());cco.d();}
      }
}
```

)

13.3.4 The `gc()` command

In addition a run-time object can be created and gargage collection can be initiated by means of the method call `gc()` to that object. JCF105 = JCFbintreebuilder ∪ `file:c.java`(

```
import tjvap.*;

class c {
static void m() {Runtime r = Runtime.getRuntime();
      {binTree x14 = binTreeBuilder.b14();cco.aw();
       soLo y14 = new soLo(binTreeBuilder.b14());cco.a();}
      r.gc();

      {binTree x17 = binTreeBuilder.b17();cco.bw();
       soLo y17 = new soLo(binTreeBuilder.b17());cco.b();}
      r.gc();

      {binTree x17 = binTreeBuilder.b17();cco.cw();
       soLo y17 = new soLo(binTreeBuilder.b17());cco.c();}
```

```
        r.gc();

        {binTree x18 = binTreeBuilder.b18();cco.dw();
         soLo y18 = new soLo(binTreeBuilder.b18());cco.d();}
        r.gc();
        }
}
```

) produces the same error once more:

```
a a
b b
Exception in thread "main" java.lang.OutOfMemoryError
```

Again the conclusion can be drawn that automatic garbage collection is not easy to predict, and not easy to provoke.

13.4 Garbage collection activated

Names local in a method body make a heavier drop out of scope after exiting from that body than by simply leaving an internal block inside a method body. At last, the garbage collection process can be activated this time! A larger tree is constructed as a consequence. JCF106 = JCFbintreebuilder ∪ `file:c.java`(

```
import tjvap.*;

class c {
static void m() {Runtime r = Runtime.getRuntime();
      m14();r.gc();
      m17();r.gc();
      m17();r.gc();
      m18();r.gc();
      }
static void m14() {binTree x14 = binTreeBuilder.b14();cco.aw();
       soLo y14 = new soLo(binTreeBuilder.b14());cco.a();}
static void m17() {binTree x17 = binTreeBuilder.b17();cco.bw();
       soLo y17 = new soLo(binTreeBuilder.b17());cco.b();}
static void m18() {binTree x18 = binTreeBuilder.b18();cco.dw();
       soLo y18 = new soLo(binTreeBuilder.b18());cco.d();}
}
```

) produces the same error, after having produced more trees, however:

```
a a
b f b
b f b
d f Exception in thread "main" java.lang.OutOfMemoryError
```

13.5 Spontaneous garbage collection

It is relevant to consider the effect of the garbage collection command by removing it from the code of the previous example. JCF107 = JCFbintreebuilder ∪ `file:c.java`(

```
import tjvap.*;

class c {
static void m() {
      m14();
      m17();
      m17();
      m18();
      }
static void m14() {binTree x14 = binTreeBuilder.b14();cco.aw();
       soLo y14 = new soLo(binTreeBuilder.b14());cco.a();}
static void m17() {binTree x17 = binTreeBuilder.b17();cco.bw();
       soLo y17 = new soLo(binTreeBuilder.b17());cco.b();}
static void m18() {binTree x18 = binTreeBuilder.b18();cco.dw();
       soLo y18 = new soLo(binTreeBuilder.b18());cco.d();}
}
```

) delays its garbage collection activities, producing fewer trees but a more complete error message:

```
a a
b b
f b b
f Exception in thread "main" f java.lang.OutOfMemoryError
        at binTreeBuilder.b02(Compiled Code)
        at binTreeBuilder.b03(Compiled Code)
        at binTreeBuilder.b04(Compiled Code)
        at binTreeBuilder.b05(Compiled Code)
        at binTreeBuilder.b06(Compiled Code)
        at binTreeBuilder.b07(Compiled Code)
        at binTreeBuilder.b08(Compiled Code)
        at binTreeBuilder.b09(Compiled Code)
        at binTreeBuilder.b10(Compiled Code)
        at binTreeBuilder.b11(Compiled Code)
        at binTreeBuilder.b12(Compiled Code)
        at binTreeBuilder.b13(Compiled Code)
        at binTreeBuilder.b14(Compiled Code)
        at binTreeBuilder.b15(Compiled Code)
        at binTreeBuilder.b16(Compiled Code)
        at binTreeBuilder.b17(Compiled Code)
        at binTreeBuilder.b18(Compiled Code)
```

```
        at c.m18(Compiled Code)
        at c.m(Compiled Code)
        at s.main(Compiled Code)
```

13.6 Memory leaks

The virtue of automatic and system-supplied garbage collection is its robustness against memory leaks. The collector should remove all garbage. The following JCF is a very limited test of that matter. JCF108 = JCFbintreebuilder ∪ `file:c.java`(

```
import tjvap.*;

class c {
static void m() {boolean b = true;
      while(b){m17();cco.c();}
      }
static void m17() {binTree x17 = binTreeBuilder.b17();cco.bw();
       soLo y17 = new soLo(binTreeBuilder.b17());cco.bw();}
}
```

) carries on and on, the garbage collector removing redundant space all the time. The generated output reads:

```
b b c
b b c
f b b c
f b b c
f b b c
f b b c
f b b c
.......
```

13.7 What is garbage?

The following JCF poses a little challenge regarding the definition of garbage. JCF109 = JCFpd ∪ `file:c.java`(

```
import tjvap.*;

class c {
static Garb g;
static void mkGarb() {
      Garb x = new Garb();
      x.init(x);
      x.get();
```

```
      bcop.p(x.equals(g));
      }
static void m(){
      boolean b = true;
      while(b){
         mkGarb();}
      }
}

class Garb {
Garb self;
void init(Garb s) {self = s;}
public void finalize() {
      c.g = self;
      bcop.pw(this.equals(c.g));
      bcop.pw((c.g).equals((c.g).self));
      cco.f();}
public void get() {c.g = self;cco.gw();}
}
```

). Its non-terminating output reads:

```
.................
4705  g true
4706  g true
4707  g true
4708  true true f
4709  true true f
4710  true true f
.................
9411  true true f
9412  true true f
9413  true true f
9414  g true
9415  g true
9416  g true
.................
```

The interesting point about this case is that the `finalize` method puts the identity of the object being collected in the variable `c.g`. Although there is no reference to it, its reference will be distributed by `finalize`. In this case the very execution of `finalize` brings the object back to non-garbage status. Then it will not be collected until the reference to it has disappeared. Subsequently it can be collected, this time without a second execution of `finalize`.

Chapter 14

Software mechanics

Empirical semantics can be used as a tool for teaching programming languages. In the case of JavaCck, the teaching efforts can be part of a broader effort to teach Java. It is also possible to include the material in teaching activities centered on the basic sequential features of object-oriented programming.

Empirical semantics and FHN can play a role in planning research work as well. The teaching of Java and program notation research are addressed independently below. The discussion of the possible use and relevance of empirical semantics will be cast in the broader context of software mechanics.

14.1 Software mechanics

Software mechanics is the theory of software construction techniques. Software mechanics deals with the functionality of the constructions rather than with their rationality. Software mechanics is preliminary to software analysis and to the implementation part of software development. Pure software mechanics deals with existing and conceivable program notations, conducted as part of pure science. Not being linked directly to ambitions of a practical nature, software mechanics is a more likely subject for pure investigation than software analysis or software design. Software mechanics is interesting in its own right and can be attributed a (limited) cultural status. Understanding software mechanics or parts of it is neither equivalent to, or a prerequisite for, the ability to program. A sound knowledge of software mechnics is considered a fruitful asset for someone who intends to perform programming him/herself.

14.1.1 Four perspectives on software mechanics

Following [5] the subject of software mechanics will allow a segmentation into four layers, each producing its own perspective. The levels are:

Theoretical software mechanics: TSM

TSM contains reflections about software construction mechanism as well as informal discussions of data gathered in the experimental part of the field (ESM below). TSM is not mathematical in style but ought to be rigorous in method. The reading and writing of language standardization documents may be considered part of TSM. That task requires a significant maturity in the field. The mathematical rigor of MSM prevents mistakes, the TSM approach requires knowledge and skill, if misjudgements are to be prevented. TSM may be considered a current de facto standard in the professional field.

Mathematical software mechanics: MSM

This subject is commonly known as (mathematical) semantics of programming languages and systems. MSM is a large and growing subject. There are many different styles of MSM. The technical part of the program algebra of [3] belongs to MSM.

The major weakness of MSM is connected with the problems in understanding it poses for programmers lacking a formal education. At the same time that is also its strength. Not being widely distributed, the formal methods of MSM are hardly 'corrupted'. These methods are indispensible if very high degrees of reliability are to be achieved.

Experimental software mechanics: ESM

ESM includes the production of empirical-semantic descriptions of existing languages, as well as the (experimental) approach to the design of novel features. Writing this empirical semantics report can be classified as ESM, reading it is best classified as ESM, unless an attempt is made to render the underlying concepts explicit. Then the work is lifted to TSM or MSM, depending on its style.

Practical software mechanics: PSM

PSM tackles the subject via the large existing volume of rational and effective software solutions to familiar problems. Most courses on programming, in particular the books on programming in Java, adopt this style in the teaching of the relevant software mechanics. We will comment in more detail on the teaching of programming in Java below.

14.1.2 Four kinds of reasoning

Together with these four forms of dealing with software mechanics come four forms of reasoning, each primarily attached to one of the perspectives.

TSM: constructive reasoning,
MSM: deductive reasoning,

ESM: inductive reasoning,
PSM: productive reasoning.

Each of these forms of reasoning has an important role to play both in the science of computer programming and in computer program engineering.

14.1.3 Pragmatic science of computer programming

One may say that science promotes competence in order to acquire knowledge and that engineering promotes knowledge in order to enhance competence.[1] This distinction between science and engineering is clear but hides an important aspect. The engineering viewpoint of knowledge need not be based on engineering as its dominant objective. The 'ability' of knowledge to enable competence is an intruiging matter in itself. With 'pragmatic science of computer programming' (PSCP) we denote the study of 'knowledge about computer programming' in its functional role of competence enhancement, the knowledge itself being the primary focus. Empirical semantics of object-oriented programming as a form of software mechanics is considered part of the pragmatic science of computer programming. The justification of this classification is easy: it probably is justified, and if it is not, that is even more interesting!

As the best possible option the authors propose to include the empirical semantics of important program notations (including the object-oriented aspects of Java) in PSCP. Lacking a long-standing confirmation of this classification, we only give an opinion. Obviously PSCP can vary in time, allowing empirical-semantics-based software mechanics to be removed if new views on programming competence emerge or acquire prominence.

Being a representative of ESM, empirical semantics relates to one's inductive reasoning abilities. It is not at all obvious how and why people are able to integrate a collection of examples, and to generate from there reasonable guesses about different and new examples. This ability is as hard to explain and as important as the human abilities for image processing. The very fact that no mathematical understanding of inductive reasoning capabilities has emerged is no argument against having a teaching method for software mechanics based on precisely that ability.

[1] In Dutch: wetenschap: kunnen om te weten, versus 'engineering': weten om te kunnen.

14.2 Teaching the software mechanics of Java

The time has come to present some explicit views on the teaching of Java[2], clarifying the potential role of this particular empirical semantics project report.

14.2.1 General remarks on the teaching of Java

It is not unlikely that teaching Java is becoming one of the very large teaching operations of this time. Java may be heading towards replacing PASCAL as a vehicle for teaching programming and the concept of programming languages, towards replacing C++ as a vehicle for library organization, towards replacing HASKELL as a tool for functional-styled programming and towards replacing Ada as a tool for concurrent programming. This implies that Java may well become a language that is taught to more people than any previous language in the history of computing. This justifies a keen interest in how the teaching of Java works, and what options it offers. Here are some scattered remarks/observations on that matter.

1. Java is usually taught to newcomers by means of a sequence of plausible examples, explaining how the language features can be of use for legitimate software production purposes. This conclusion can be drawn from a survey of the abundant introductory literature on Java. Attempts to provide a complete and precise syntax in combination with clear semantic insights are remarkably rare.

2. The teaching of Java mainly exploits the inductive reasoning capacities of the audience, expecting them to make reasonable guesses concerning rational class writing from a sound body of examples.

3. Teachers of Java differ vastly in the perception of 'the essence of Java'. Not even a helpful classification of the various positions is easy make.

4. Java being a changing language, focussing one's attention on problematic aspects is a potential waste of time, these aspects being candidates for improvement all the time.

5. Java has the size of an operating system like UNIX rather than the limited scope and syntax of ALGOL60. Knowing 'all of Java' is an objective as little reasonable an objective as it is to know everything about UNIX.

6. The literature is utterly inconsistent regarding simple questions like: for which software production tasks is Java best suited, for which sofware production problems may Java be considered the best possible solution, why

[2] Java is sufficiently important to concentrate a formidable teaching effort world-wide. Java may be considered a land-slide victory for imperative programming. It proves that designers of imperative languages are able to design a program notation which is good-looking and disappointingly slow on the computer at the same time. The designers of Java have thus made most non-imperative methodologies look almost ridiculous. Indeed, it is hard to believe that logic programming is an option if seemingly trivial Java compilation and execution still poses formidable problems.

has Java become so important, is the Java design a scientific achievement, can the language be separated from SUN's implementation ideology (using standardized bytecodes and the JBVM), how is Java related to software component technology, is a class file a software component, what is meant by the Java security model, and why does it work? It may be argued that these questions pertain to a level of abstraction above the individual program notation.

7. Java is a matter of fashion and style, just as much as it is a technical tool. In the capacity of a carrier of fashion and style its borders are fuzzy everywhere. This fuzziness is sometimes driven to its extreme: books on Java Beans may not even bother to spend a single line on the question 'what is a Java Bean'.

8. The class writing ideology takes the aforementioned unclarities for granted by aiming at getting acquainted with a sequence of program notations and techniques on a purely technical and formalistic basis. Class (family) writing and class (family) reading are potentially instrumental ingredients of this familiarization process.

9. It is implausible to expect a principled approach to programming-language concepts and structures to produce a meaningful rationale for an ad hoc notational conglomerate such as Java. The unprecedented popularity of the language cannot change that fact. If the principled programmer or software researcher sets his or her feet on the Java training avenue, he or she cannot but adopt an example-driven style. The present class-writing ideology, together with an eclectic style of sublanguage picking, allows the principled teacher to work without being forced to semantic pedantry in a way entirely out of tune with the working modes of the designer community having invented Java in the first place.

14.2.2 The teaching of Java and empirical semantics

Emperical semantics can play a role in the teaching of the Java software mechnics. The following remarks refer to possible teaching practices:

1. Knowledge of JavaCck and its empirical semantics implies the ability to compute by hand the outputs of a correct (and sufficiently simple) JCF. Most examples in the previous chapters are of that nature. It is reasonable to expect students to be able to do so without access to this text, an 'open book' strategy being equally reasonable.

2. A written exam would simply require the student to demonstrate this ability within a limited amount of time. The document can be viewed as a series of exercises, the aim of the exercises being the correct prediction of the correct JCF output.

3. The text is probably best used in the absence of any computer or implementation. By viewing the matter as an entirely theoretical issue the fastest track to an understanding of the basics of sequential object orientation is found.

4. The empirical semantics of JavaCck can be understood and taught from first principles to an audience without any previous exposure to programming notations. Rational programming (class writing serving a purpose in a rational way) can subsequently be taught. For the task of rational programming an awareness of the empirical semantics of the used program notation is a useful prerequisite.[3]

5. In this text we attempt to facilitate the teaching of aspects of Java as if the language and its implementation were completely stable, assuming that it will be prominent for a long time to come. Of course, we cannot predict the future of programming. But, if computer programming were to stabilize for a fairly long period in the future, along lines quite close to Java: what would its initial teaching look like? We present empirical semantics as an option.

14.3 Compiler dependency: weakness or strength?

Whereas software mechanics may be regarded as a basically compiler-independent topic, empirical semantics as a perspective on software mechanics is not.

Empirical semantics is patently compiler-dependent. It observes compiler behavior without any prejudice. Future compilers will be different and so will the observed empirical semantics of the various features. At first sight, this seems to constitute a weakness of empirical semantics. We would like to argue against that conclusion. The key observation is that if a compiler-independent teaching of a language is deemed important (which we will never deny), it should be put into practice in a realistic manner. There seems to be no sign of such a tradition around. The efforts needed to teach the mathematical preliminaries required for a compiler-independent introduction to programming language semantics are usually considered prohibitive. To put it even more strongly, the very ambition to obtain compiler-independent descriptions through formal means is often considered 'riding one's hobbyhorse'. No other techniques to arrive at compiler- independent descriptions have surfaced, however. In a world

[3] Rational programming comes in many styles and brands. The designers of computer science curricula seem to adopt the view that 'having some programming style' is better than 'having no style at all'. As a result, programming styles are often included in the curricula, without them being compared in terms of programming objectives, and without any evidence that teaching programming styles is conducive to a better understanding of the features of a programming language.

Empirical semantics, however, is far more limited and predetermined than teaching a programming style. Indeed, one may even decide not to teach rational programming in a preferred style, and to restrict teaching to the subject of empirical semantics. This is a very extreme position, and it only makes sense if active experimentation is part of the courses.

that rates the cost of acquiring compiler-independent mathematical precision too high, the compiler 'is the meaning'. This is an entirely legitimate viewpoint, if only its consequences are accepted as well.

Empirical semantics takes that viewpoint as a point of departure. Thus, it accommodates to a deep psychological need for 'intuitive understanding' while not doing away with elementary scientific principles. Needless to say, the price paid for this 'soundness' is superficiality. But, why should empirical data be 'deep'?

14.3.1 Incremental feature understanding

Features such as CI, CE, CAS or THI can only be understood within a context of other features. We have given an expanding sequence of feature collections and particular features have been revisited in ever expanding contexts. There is no such thing as 'knowing what an object is'. Individual features cannot be understood once and for all. Every understanding of a feature is necessarily restricted to a context, the impact of context extension generally being totally unpredictable.

JavaCck not only constitutes a limited set of features but also a limited context. Understanding the empirical semantics of various features in JavaCck is not the same as understanding the role of the same (syntactic) features in larger subsets of Java. The introduction of multi-threading, for instance, forces one to refine the notion of an object, taking an underlying phase space into account as well.

One may question the usefulness of a limited understanding of features. In reply it may be argued that, program notations being in a constant state of flux and development, a context-dependent understanding is always the best one can get. Being aware of that context is more important than working in the largest currently available context. Indeed, leaving the context implicit forces one to always work in the latest framework. This does not seem feasible even for industry. A fortiori, it is not feasible for academic teaching and reasearch.

14.4 Research on empirical semantics

Software mechanics (SWM) is a substantial subject, growing in size every day. Taking empirical semantics as a subtopic of SWM means limiting its future by that of SWM.

It is evident that many more reports can be written along the lines of this document. Furthermore, the selection of cases can be based on a far more systematic approach.

14.4.1 Proceeding to TSM and MSM

Another forward track for research in empirical semantics is to use it to organize work on formalized semantic models. Given an empirical semantics report, a

formalized semantics can be defined as a consistent logical theory able to produce predictions for JCF behavior, and in particular able to predict the body of data contained in the empirical semantics report. Besides being consistent, a formalized model should be complete. This means that it produces behavioral predictions for all possible syntactically correct programs.

It is an intruiging question to what extent AI-techniques can be applied to automate the transformation from an empirical semantics report to an informal but complete language description (compatible with TSM) or even to a formal description (acceptable for MSM).

14.4.2 The psychology of the mental picture

A mental picture of single-threaded, single-inheritance-based object-oriented program notations is supposed to emerge from reading this document in detail. We have little specific information about this mental picture. Many questions can be formulated, some of which may be of practical relevance.

1. To what extent are people aquainted with this text able to predict the behavior of 'similar' programs. Is there a kernel of JavaCck for which the picture will be very good, slowly or even steeply degrading outside that kernel?

2. Is this document informative for readers with a formal/mathematical inclination, or is it disappointing and frustrating for these readers by being utterly incomplete and inconclusive?

3. Is this empirical semantics for object-oriented syntax an invitation for further investigation or does it leave a confusing and somewhat disappointing impression?

4. Is formality at all a useful direction for establishing an improved mental picture concerning the core features of object-oriented programming?

5. Do people with a top ability to predict program behavior (in the range of examples as displayed above) use a mathematical/formal model, or do they use intuitions that have grown by experience?

6. The previous question in a more speculative form: suppose JavaCck Competitions were organized on a regular basis, requiring participants to produce program behavior predictions. What preparation would be developed for ambitious participants (formal methods and projection semantics, compiler studies or compiler writing, massive programming experience) ?

7. Can results similar to de Groot's famous findings on the modes of thinking of chess players be established for programmers. If so, will it turn out that strong Java programmers are better in predicting program behavior for programs like the ones above than the average programmer?

8. Should it be expected that programmers only use program constructs with which they are familiar, leaving other features unused because of a lack of confidence or appreciation?

9. Will the exposure to a large collection of features and mechanisms increase the probability of them being used by a programmer? Are there any features which a reader of our empirical semantics is more likely to use than a programmer having been exposed to more traditional introductions to object-oriented programming in Java?

14.4.3 Semantic conjectures on JavaCck

The empirical work on JavaCck suggests several global semantic properties of Java that may merit further investigation. We formulate these as conjectures for the simple reason that no proof is known to us, however clear the matter may be from an intuitive point of view. (Of course the conjectures may also be wrong and even be known to be wrong. Proving such a conjecture (in the setting of empirical semantics!) amounts to the following: (i) providing a mathematical model of the semantics of the language which is consistent with the empirical data, (ii) proving the conjectures in that setting.

Even after having been proven the conjectures may be refuted by experiment. The proof is nothing more than an argument inside a formalization consistent with a given finite body of (empirical) knowledge.

1. Without the use of sockets or pipes and without the use of thread creation and without using the `wait()` action no programs can be written that result in a deadlock when being executed. (the use of synchronized methods is allowed of course).

2. Definitive method overriding: suppose X is an object of class A, class B is a superclass of class A, both A and B have a method m() and the respective method bodies differ: let C be some new class with a static void method f(B x). The conjecture is that, if f is applied to X, inside the body of f the body that m() has in B cannot be activated. (More precisely: there is no general way to activate the overridden methods of X 'from outside'.

3. Casting limits. Let X = Exp be an assignment in a JCF. Suppose that replacing X = Exp by X = (C) Exp, for an appropriate class C, gives a syntactically correct program. In that case its functionality equals that of the original program (outputs must be identical).

4. Minimal separable pair. The pair found in 9.9.4 is minimal in the sense that there are no shorter pairs of classes that can be distinguished by adding them to identical contexts (sum of text lengths).

Bibliography

[1] K. Arnold and J. Gosling. *The Java Programming Language.* Addisson-Wesley, New York, 1996.

[2] J.A. Bergstra, I. Bethke, and P. Rodenburg. A propositional logic with 4 values: true, false, divergent and meaningless. *Journal of Applied Non-Classical logics*, 5:199–217, 1995.

[3] J.A. Bergstra and M.E. Loots. Program algebra for component code. Technical Report P9811, Programming Research Group, University of Amsterdam, 1998.

[4] J.A. Bergstra and A. Ponse. Process algebra with five-valued conditions. In C.S. Calude and M.J. Dinneen, editors, *Combinatorics, Complexity, and Logic, Proceedings of DMTCS'99 and CATS'99.* Springer-Verlag, Singapore, 1999.

[5] J.A. Bergstra and S.F.M. van Vlijmen. *Theoretische Software-Engineering.* ZENO-Institute, Leiden, Utrecht, The Netherlands, 1998. In Dutch.

[6] M.A. Ellis and B. Stroustrup. *The annotated C^{++} reference manual.* Addisson-Wesley, New York, 1990.

[7] J. Hunt. *Java and object-orientation, an introduction.* Springer, 1997.

[8] B. Jacobs. A counter-example to PCA, 1999. Personal communication.

[9] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice Hall, 1978.

[10] P. Naughton and H. Schildt. *Java 1.1, the complete reference.* Addisson-Wesley, New York, 1998.