Qualitative analysis of Mastery Checks in a programming course

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics

presented by
Luca Chiodini

under the supervision of
Prof. Matthias Hauswirth

July 2020

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Luca Chiodini
Lugano, July 2020

To those who suffer in this pandemic
and to those who soothe pain

Abstract

Learning to program is hard. During the last decades, a number of pedagogical approaches to improve the teaching of programming have been proposed, with different degrees of success. The ultimate goal of teaching effectiveness is to reach the level one can obtain with one-to-one tutoring without resorting to it.

We carried out this work in the context of a first-year second-semester programming course held at USI to teach students about object-oriented programming in Java, after having learnt functional programming. The course uses a variation of the mastery learning approach originally devised by Bloom and adapted to programming by Wrigstad and Castegren at Uppsala University.

The pillar of this thesis is a qualitative study conducted with six students; we recorded their performance during ten mastery check sessions aimed to assess their understanding of all the fundamental topics in an object-oriented introductory programming course.

For this purpose, I developed a new tool to automatically integrate videos from two cameras, the screen recording and a transcription obtained with a privacy-preserving mechanism. Then, we coded all sessions using an open coding approach with MAXQDA, a leading software for qualitative research.

Finally, we discuss some insights enabled by the rich set of information uncovered in the study. They cover the misconceptions developed by novices related to the programming language and, perhaps even more importantly, to the strategies used to tackle problems and to the issues that arise when using the notional machines introduced in the course. Moreover, exploiting the temporal sequence of the ten sessions, some hypotheses are formulated about the learning trajectories and the impact of teaching interventions on the persistence of misconceptions.

This in-depth microgenetic qualitative study is, to the best of our knowledge, the first attempt at capturing and analysing interviews with students without being limited to closed-form answers. Its realisation makes possible to conduct in the future more targeted studies to validate hypotheses and diagnose specific issues highlighted in state-of-the-art computer science education research.

Acknowledgements

The first thought goes to my parents, who provided me with everything I needed up to this point in life: they made possible for me today to write this thesis.

My deep gratitude goes to Matthias Hauswirth, my supervisor, who first introduced me to the world of computing science education in which I was always interested but never did concrete steps towards it. He has constantly encouraged me in the past months that have been exceptional, to say the least. I am indebted to his continuous availability even among all other duties: it really mattered a lot.

I would also like to thank my friends, both the ones who shared with me this double degree programme between Milan and Lugano and those who live in Italy and I occasionally had still the pleasure to meet and talk with. I may not always show excitement, but I love the small moments we spend together.

Thanks!

Contents

Contents ix

List of Figures xi

List of Tables xiii

1 Introduction 1

1.1 Learning to program . . . 1
1.2 Thesis organisation . . . 3
1.3 Contributions . . . 3

2 Mastery Checks as a way of assessing knowledge 5

2.1 Mastery Learning . . . 5
2.2 Mastery Learning in programming courses . . . 6
2.3 USI's Programming Fundamentals 2 course . . . 7
2.4 Notional Machines used in the course . . . 11
2.4.1 Stack and Heap diagram . . . 12
2.4.2 Sequence diagram . . . 13
2.4.3 Expression tree . . . 15
2.4.4 Control-flow graph . . . 17
2.5 Programming misconceptions . . . 18
2.6 Mastery Checks for research . . . 21

3 Designing the qualitative study 23

3.1 Microgenetic method . . . 23
3.2 Research questions . . . 24
3.3 Ethical and privacy issues . . . 24
3.4 Student recruitment . . . 25
3.5 Technical aspects . . . 25

4 Mastery Check sessions 31

4.1 Session 1: Classes versus Objects and Method implementation . . . 32

4.2 Session 2: References and Stack and Heap 34
4.3 Session 3: Method invocation and Sequence Diagrams 35
4.4 Session 4: Control Flow and Conditional Computation 37
4.5 Session 5: Recursive Computation and Iterative Computation 38
4.6 Session 6: Recursive data structures and Variables 39
4.7 Session 7: Literals, Types and Expressions 40
4.8 Session 8: Inheritance and Polymorphism 41
4.9 Session 9: Use of generics, ArrayList versus array 43
4.10 Session 10: Abstract classes and Interfaces 44

5 Coding in MAXQDA 47
5.1 A-priori versus open coding 47
5.1.1 K-Java rules as codes 48
5.1.2 Parsing to discover codes 49
5.2 Codes 50

6 Insights about misconceptions and strategies for solving problems 57
6.1 Several misconceptions are related to the programming language 58
6.2 Some misconceptions are one the dual of the other 60
6.3 Misconceptions can be caused by wrong analogies 64
6.4 Notional Machines can also harm 68
6.4.1 Students improperly use Notional Machines 68
6.4.2 Notional Machines are incomplete 72
6.5 Tackling a problem the right way is hard 76
6.5.1 Recursion problems are especially hard 78

7 Insights about learning trajectories 85
7.1 Misconceptions persist over time if not corrected 85

8 Conclusions and follow up studies 91

Bibliography 93

Figures

2.1 Stack and Heap diagram notation. 13
2.2 Sequence diagram notation. 15
2.3 Expression tree notation. 17
2.4 Control-flow graph notation. 19

4.1 Stack and Heap diagram produced by a student during the first mastery check. 33
4.2 Control Flow Graph for the method `firstChar` produced by a student during the fourth mastery check. 38
4.3 Sequence diagram produced by a student during the fifth mastery check. 39
4.4 Expression tree produced by a student during the seventh mastery check. 41
4.5 Stack and Heap diagram produced by a student during the ninth mastery check. 44

5.1 A screenshot that shows how MAXQDA looks like. 48
5.2 Rule for `try` in K-Java. 48

6.1 Map of codes about misconceptions related to the Java language. 59
6.2 Map of dual misconceptions. 60
6.3 Map of misconceptions caused by wrong analogies. 65
6.4 Map of codes related to notional machines. 69
6.5 Map of codes related to how to tackle problems. 76

Tables

2.1 Mapping course weeks, book chapters and Java topics. 11

4.1 Mapping course weeks, mastery check sessions and Java topics. 32

7.1 Correctness of ThisExistsInStaticMethod across four sessions. 87
7.2 Correctness of SuperclassObjectIsAllocated across two sessions. 88
7.3 Correctness of StringLiteralInsideLocalVariableInStack across two sessions. 89

Chapter 1

Introduction

1.1 Learning to program

Learning to program is hard. While an ever-increasing number of people are trying all over the world to master the art of programming, there is plenty of evidence [Guzdial, 2015] that the job is not trivial. The process that starts with one being unaware of what programming even means to being an effective developer is known to be long and difficult for many.

In 1986 [Soloway, 1986], Soloway proposed the so-called “rainfall problem”, a simple programming task whose original version can be summarised very quickly.

> Write a program that will read in integers and output their average. Stop reading when the value 99999 is input.

Teachers and professors expect their first-year students to be able to solve this relatively straightforward task by the end of their introductory programming course. Soloway, who taught at Yale, one of the most prestigious universities in the world, found that after the first term just 14% of students were able to come up with a correct solution after being given a reasonable amount of time to solve the problem. That percentage increased only to 36% after the second term and to 69% at the end of the second year. The facts that not everybody can reach a solution without external help after just having programmed a semester and that the share of successful students increases as time progresses are two reasonably expected outcomes; nonetheless, what catches the eyes is that the striking majority of students (86%) could not solve the task after completing the first course.

Even though one might be tempted to say that those results were due to improper measurements, a wide body of literature has shown that the pattern repeats with little differences throughout different universities. A 2001 study [McCracken et al., 2001] carried out with 216 first-year students from four different institutions in different na-

tions reported an average score of 22.89 out of 110 points when confronted with problems similar to the rainfall one.

In 2013 [Utting et al., 2013], the same kind of study was replicated with an even larger setup which included 418 first-year students across 12 universities located in 10 different countries. Students were divided into two groups: one was provided with significant scaffolding and a test suite to ease the development while the other one did not have any kind of help, as in the previous studies. Results have shown that performance ranges between 12% to 32% for those without scaffolding and from 61% to 75% for those who had it.

At this point, it is unfair to claim that novices' poor results represent specific situations where something does not work and someone can be blamed (either students or teachers). We should instead acknowledge that, besides the fact that learning in general is not easy, learning to program has its own intrinsic difficulties. Significant research should be carried out to investigate the reasons why this activity often is unapproachable by many, to understand which errors are widespread, and to devise new teaching methodologies. All of this is the scope of the field of informatics known as computer science education (CSE).

From psychology and pedagogy theories, we learn many insights about how cognitive processes develop and when we observe changes. Stern [Stern, 2005] says that "the precondition for top performance in fields like chess, music or physics is a very extensive base of concrete situational knowledge". That is naturally also the case for computer science. Conceptual knowledge is "usually viewed as general and abstract knowledge of the core principles and their interrelations in a domain" [Schneider and Stern, 2010].

In this work, we will delve into a first-year programming course held at Università della Svizzera Italiana (USI) which is focused on object-oriented programming and uses Java as programming language. The bigger study of which this work is part of consists of two phases: an exploratory stage and an experimental one.

This work constitutes a significant part of the exploratory stage. We want to capture the performance of a selected number of students throughout the course, analysing their answers to various kinds of questions asked during weekly mastery check sessions. These sessions are recorded and later undergo fine-grained qualitative analysis to elicit information about common misconceptions, different programming strategies, errors recurring in time and knowledge development. We will formulate hypotheses on pedagogical interventions that might be beneficial to students, like improvements on the notional machines which are used to assist teaching during the course, highlights on the most difficult topics, points that greatly suffer from the "expert blind spot" and any other kind of useful advice.

The experimental phase, which is out of the scope of this work but it is essential to validate the hypotheses, will consist of controlled experiments informed by the rich set of insights gathered during the previous phase. Since in general it is extremely tough

to measure the degree of effectiveness of a pedagogical intervention, these experiments will probably be focused on a single aspect to corroborate our findings.

The ultimate goal of the project is to answer several open research questions in computer science education. In 2009, Sheard states in his analysis on the status of research into the teaching and learning of programming that “it is broadly recognised that programming is hard to learn and therefore hard to teach well” [Sheard et al., 2009]. We want to contribute to this, trying to inform teachers better so that both them and students can teach and learn programming more effectively.

1.2 Thesis organisation

This thesis is organised in chapters. Chapter 2 presents the idea of mastery learning, which is not exclusive of computer science, and mastery checks as a way to assess knowledge; it also contains an overview of the course tied to this study and a summary on the previous work about programming misconceptions.

Chapter 3 provides a detailed view of all the facets of this research qualitative study. It describes everything one needs to take into account to set up a study like this one, starting from the research questions to the ethical issues. It ends with a rich description of the technical aspects, which required to create a new piece of software to assist the research.

Chapter 4 lists, for each of the ten sessions of this study, the topics checked during the mastery session and the questions asked to the students, along with reasons for doing so. All the sessions are then carefully analysed following the methodology described in Chapter 5, which outlines and briefly comments on the codes used to tag relevant segments of video recordings.

Chapters 6 and 7 constitute the core and the outcome of the study; they describe insights we have discovered after a throughout analysis of the codes. We present misconceptions and strategical mistakes in chapter six, accompanied by interesting examples directly extracted from the sessions. In chapter seven, instead, we discuss the learning trajectories, focusing on the same misconceptions across multiple sessions.

Lastly, Chapter 8 is used to draw conclusions on this qualitative study and start the discussion on what can be done in the many possible follow up specific studies in the experimental stage.

1.3 Contributions

These are the key contributions of my work:

- I developed a tool that integrates multiple video sources, automatically selecting the interesting segments, and that captions an interview keeping data offline at

all times. It can be used in other research contexts where interviews constitute a key part of the “data collection” stage.

- I recorded ten individual mastery check sessions on object-oriented programming topics with six students, asking questions and observing the answers and the process they took to get there. Recordings are available for future studies, in compliance with the informed consent freely signed by each participant.

- I analysed all the recorded sessions to track and understand which misconceptions students have, which strategies they use to solve different kinds of problems, and what are their learning trajectories. I provide examples with artifacts produced by participants and I try to classify all the above into sensible categories.

Chapter 2

Mastery Checks as a way of assessing knowledge

2.1 Mastery Learning

More than fifty years ago, in 1968, Bloom introduced in a famous paper the concept of "learning for mastery" [Bloom, 1968], also called "mastery learning". He argued that the vast majority of students, perhaps over 90 percent, is theoretically able to master the contents they are supposed to learn and, if they do not reach this goal, the problem must reside on the yet unexplored ways to properly and effectively teach a subject. The key observation is that although each individual has a different learning rate and thus needs a certain amount of time to master something, if teachers could devote that time in appropriate learning conditions, the student would almost certainly be able to attain mastery.

The strategy in Bloom's mastery learning approach requires firstly to organise and group concepts into topics, themes, units or whichever form works better. The idea is that each chunk should be taught and studied roughly in one week. Then, students are required to demonstrate to have acquired mastery on that specific topic through some kind of formative assessment procedure that should be aimed at identifying and characterising the status of knowledge of that student for that particular topic. Feedback and corrective measures, again targeted at that specific topic, should at this point be put in place to guide the student and fix possible errors or holes in the knowledge. When a formative assessment is passed, we say that the student has achieved mastery on the topic.

Guskey notes [Guskey, 2010] that there are several components that should not be overlooked in the mastery learning approach. It is important to start very early and possibly even before the beginning of the actual course with a preliminary formative assessment to understand what students actually know and do not know before taking the course. In fact, it has been shown [Ambrose et al., 2010] that missing preliminary

knowledge may severely impede learning and this is exacerbated when there is a misalignment between what the teacher thinks students know and what students actually know.

Assessments themselves represent a key part of the process. It is crucial to hold them regularly so that they are tightly coupled with the course content and can serve best their purpose to understand what is the “state” of the classroom. But, maybe even more importantly, high-quality feedback should immediately follow. It has to be timely, as leaving things in their bad shape would be detrimental for the progress in the course; it does need not to be another session of teaching nor a piece of generic advice sent to students. On the contrary, high-quality feedback means that is given specifically for that student, accommodating learning differences, and is tailored to the specific errors made or the revealed misconceptions.

2.2 Mastery Learning in programming courses

In recent years, the possibility to apply mastery learning in the context of programming courses has been explored. One of the first documented approaches comes from Uppsala University, Sweden [Wrigstad and Castegren, 2017]. The authors are responsible for a second-year, large (more than a hundred students) and big (20 ECTS) programming course where they teach imperative and object-oriented programming.

The course has been restructured in 2013 with the goal to improve students’ outcome and have them assume more responsibility in their learning process. In this respect, they also wanted students to make informed decisions about the skills they want to acquire and the time when they want to do so. Wrigstad and Castegren developed a system called “Achievement Unlocked” which contains elements and ideas like the ones proposed in [Bloom, 1968].

Essentially, a list of all the skills that could possibly be learnt has been derived, thanks to prior experience and the ACM/IEEE Curriculum for Computer Science. Each of these skills, called achievements, has been appointed to a specific grade level. To earn a certain grade, students must demonstrate mastery on all the skills classified at that level and those below it.

This methodology has several advantages. Clearly, it makes students aware of everything they could possibly learn going through the course. What is better, though, is that students are also encouraged to reason about which subsets of those skills are truly fundamental and cannot be missed. On top of this, by forcing students to also demonstrate competency on everything belonging to lower levels even if they want to reach a higher one, it prevents the overlooking of the crucial parts to solely favour the ones classified as more advanced.

As achievements are not tied to assignments nor dates, students are effectively free to schedule their attempts to demonstrate mastery at their own pace and only when they feel to have learnt the specific subject. However, since this might lead to unde-

sirable behaviours such as passing by "brute forcing" (trying to demonstrate mastery repeatedly on the same topic hoping that eventually you will succeed), the number of available slots is limited on purpose.

Examinations take place as an oral session with a teaching assistant working as an examiner and two students: this saves human resources halving the number of required sessions but also enables interesting discussions exploiting the interactions between the students. Mastery must still be demonstrated and is still awarded on an individual basis.

Overall, the approach of [Wrigstad and Castegren, 2017] is heavily inspired by the theory of constructive alignment proposed by [Biggs, 1996]. It takes ideas from constructivism and puts emphasis on the intended learning outcomes, which in our context are represented by the aforementioned skills. Results greatly benefit when the curriculum and the assessment methods are aligned. Biggs himself notes, in the same paper, that mastery learning is a particularly interesting example and questions whether narrowing the scope of the single specific outcomes comes at the cost of broader, high-level outcomes.

2.3 USI's Programming Fundamentals 2 course

At USI, first-year students enrolled in the Bachelor of Science in Informatics are required to take the "Programming Fundamentals 1" (PF1) course during the autumn semester, during which they are introduced to functional programming using a subset of Racket[1]. Then, in the spring semester, they are expected to learn a different programming paradigm, the object-oriented one, using Java as a programming language. This second course called "Programming Fundamentals 2" (PF2) is, for students without prior programming experience, just the second time they get a chance of being exposed to programming.

The course is organised along 14 weeks, the usual duration of a spring semester excluding the Easter break, and follows closely the book "Objects first with Java" [Barnes et al., 2006]. Each course week is mapped to a chapter of the book.

As stated, a prerequisite for applying mastery learning techniques to a course is to define the learning objectives as clearly as possible and to reason about exactly which topics students are expected to learn. Over the years, USI's Programming Fundamentals 2 gained a well-refined structure which is worth explaining in detail.

The whole course content is organized as a hierarchy of 7 themes and 53 topics; themes group and contain multiple topics. Each topic is then further refined in a number of skills, usually from three to six, which are intended to exactly list what the student is required to know in order to demonstrate full mastery on that specific topic. The list of themes and topics will now follow, those marked as optional are not really part of the material taught during the 2020 edition of the course due to lack of time.

[1] Specifically, the subset of Racket used is called Beginner Student Language (BSL).

- Theme: PF1
  - Background knowledge from PF1: required knowledge expected to be learnt during the course in the previous semester
- Theme: Java Language
  - Classes vs. Objects: difference between classes and objects and analogies with other terms (model, prototype)
  - Method invocation: how to invoke instance methods and constructors, pass parameters and use the return value
  - Method implementation: use of parameters and `this`, how to return a value
  - Literals: recognise and create literals of different types (primitive types and `String`)
  - Operators: use of arithmetic, logical and relational operators
  - Types: determine the type of an expression, casting
  - Auto boxing: exploit auto boxing and auto unboxing
  - Variables: declaring a variable with a type, assignments to and readings from a variable
  - Enums: use of enumerations
  - Expressions: evaluation and type-checking of expressions
  - References: aliasing, heap allocation, null, reference equality versus value equality
  - Array basics: use of one-dimensional arrays
  - Arrays: allocation and use of one- and multi-dimensional arrays
  - Inheritance: class inheritance, subtyping and how to up and down cast
  - Polymorphism: method overriding, static and dynamic type, dynamic method lookup
  - Abstract classes and Interfaces: interfaces, abstract classes, abstract methods and how to obtain multiple inheritance through interfaces
  - Use Generics: instantiate and use generic types (Java collection classes)
  - Packages: import, division in packages
  - Static members: static methods and static fields
  - Inner classes: anonymous inner classes
  - Assertions (optional): use of assertions
  - Exception handling (optional): use of `try/catch` blocks, throw exceptions, difference between checked and unchecked exceptions, use of finally, common exceptions

  - Reflection (optional): `instance of`, advanced use of class `Class`

- Theme: Algorithms and Data Structures
  - Conditional computation: implement code that executes based on a condition
  - Iterative computation: implement code that repeats using loops
  - Recursive data structures: define classes for linked lists, trees and graphs
  - Recursive computation: implement traversal and operations on recursive data structures, understand base case versus recursive case
  - Implement Lists, Sets, Maps: write your own implementation of lists, sets, maps and define iterators over them
  - Hashing: understand the basics of hashing and use it in `hashCode` and `equals` methods
  - Regular expressions (optional): use regular expressions to do pattern matching on strings

- Theme: Java API
  - Use Lists: use of `ArrayList` and iterators
  - Use Sets and Maps: use of `HashSet` and `HashMap`
  - AWT and Swing: GUI creation using GUI-related components, implementation and registration of listeners
  - Input and output (optional): use of classes to read and write textual and binary files
  - Serialization (optional): use of classes to serialize objects

- Theme: Design concepts
  - Naming: use good names and follow code conventions
  - Information hiding and encapsulation: understand why those principles are important and argue about code that follows or violates them
  - Coupling and cohesion: understand why those principles are important and refactor code to reduce coupling and increase cohesion
  - Immutability: understand its benefits, how to make a class immutable, `final` fields
  - Abstraction: identify common parts and extract them in an abstraction layer
  - Observer pattern: understand and implement the observer pattern
  - Composite pattern: understand and implement the composite pattern

  - Visitor pattern (optional): understand and implement the visitor pattern

- Theme: Development
  - Unit testing: use of JUnit to write unit tests
  - Debugging: use of a debugger, understand stack traces, print-debugging
  - Build and running: solve compilation errors, run code inside and outside the IDE, know the `main` method
  - Javadoc: understand and write Javadoc comments

- Theme: Notional machines (this theme is explored in detail in Section 2.4)
  - Stack and Heap: draw the state of call frames in the stack and objects in the heap at a given point during the execution of a program
  - Sequence diagram: draw UML-style sequence diagram to show the sequence of method calls and returns
  - Expression tree: draw a tree to show the decomposition of an expression into subexpressions
  - Control-flow graph: draw a graph which shows the control flow of a method to explain conditional and iterative computation
  - Call tree (optional): draw a tree of method calls and returns, in alternative to the sequence diagram

To contextualise the core topics, the ones listed under the Java language theme, Table 2.1 shows the sequence of the 14 weeks of the 2020 edition of the course mapped to the chapter of the book in focus for that week and to the new topics introduced. It is important to remark, though, that a topic is not necessarily exhausted in one week and further details about it might be explained during later weeks. Note also that week 9 is absent for the Easter break and that week 15 completes the course but does not cover additional material.

This course does not adopt the extreme approach suggested in [Wrigstad and Castegren, 2017] and still uses as assessments regular exams (one midterm and one final exam) and a programming project to be developed in pairs during the second half the course. However, mastery check sessions are offered every week to consolidate students' knowledge and provide them with the opportunity to test very early their skills. Although failing or passing a check is not directly reflected into the final grade, the participation is encouraged also by rewarding students with bonus points on the "participation" component of the final grade (which accounts for 15 percent). Programming assignments, called "labs", complete this participation grade and allow students to practice writing code in Java on realistic scenarios.

Course week	Book chapter	New Java language topics
Week 1	1: Objects and Classes	Classes versus Objects, Method Invocation, Types, Method Implementation
Week 2	2: Understanding class definitions	Variables
Week 3	3: Object Interaction	Literals, Operators, Expressions, References
Week 4	4: Grouping Objects	Use of generics, Packages
Week 5	6: More-Sophisticated Behavior	Autoboxing, Static Members
Week 6	7: Fixed-Size Collections: Arrays	Array Basics, Array
Week 7	8: Designing classes	Enums
Week 8	9: Well-Behaved Objects	/
Week 10	10: Improving Structure with Inheritance	Inheritance
Week 11	11: More About Inheritance	Polymorphism
Week 12	12: Further Abstraction Techniques	Abstract classes and Interfaces
Week 13	13: Building Graphical User Interfaces	/
Week 14	14: Handling Errors	Exceptions

Table 2.1. Mapping course weeks, book chapters and Java topics.

Unfortunately, the 2020 edition of the course has been severely impacted by the coronavirus pandemic and only the first three weeks of the course took place in person, with two mastery check sessions held at the end of the second and the third week. The rest of the course was offered remotely via real-time video lectures and we eventually decided not to continue having mastery checks with the class as they proved to be too time consuming, if one accounts the overhead of scheduling remote meetings and setting up the environment.

Nonetheless, mastery checks have been used as planned to "interview" students in the qualitative study described in detail in Chapter 3.

2.4 Notional Machines used in the course

The first appearances of the term *notional machine* are dated back to the 1970s [Miller, 1974] when non-programmers started to approach programming and the first relatively simple programming languages have been developed: there was a need for methods to explain the underlying mechanisms that regulate the execution of a program. Later, du Boulay used "notional machine" to characterise "the general properties of the machine

one is learning to control" [Du Boulay, 1986]. He argued that a significant part of the difficulties that novices encounter when trying to learn programming are due to misunderstandings on what exactly this machine can and cannot perform. In general, a notional machine is thus an idealised view of a computer and the scope of what we can do with it is determined by the constructs of a programming language, both in terms of syntax and semantics.

Pedagogy has shown that it is beneficial to explicitly bring up the notional machine to *explain* how things work and, in this domain, how a program is executed. The point is to choose an appropriate *level of abstraction* at which to explain what is going on when a piece of code is being executed by the underlying real machine, the computer. Obviously, there is no one-size-fits-all notional machine, as one might want to explain concepts at different depths or from different points of view throughout the course.

Programming Fundamentals 2 at USI explicitly teaches and requires students to learn about four important notional machines which are described in the next paragraphs. We want to stress the fact that the purpose of clarifying these machines is not introducing another rigid formalism that has to be learnt, but instead providing to the students a diagrammatic way to deeply understand their own Java programs, as well as those written by others.

2.4.1 Stack and Heap diagram

During the execution of a Java program, we can identify three main memory areas: the stack, the heap and the globals. We deem the first two essential to understand what is going on at a given point of execution and for this reason we introduced a "Stack and Heap" diagram that graphically shows what their content is.

The stack is drawn on the left and contains a number of *stack frames*, one for each method that gets invoked, constructors included. As it happens in the real memory, stack frames are pushed on the top of the existing ones and are popped from the stack after the end of the execution of a method.

Each stack frame is labelled with the name of the method in the form `Class.method()` and contains all the local variables that get allocated during the execution of the method plus all the parameters. Each of these things (local variables and parameters) is represented as a small box labelled with the name of the variable or the parameter and its type, which can be either primitive or a reference. When we deal with a primitive type (an integer number or a character, for instance) we put the value right inside the box; on the contrary, for reference types we always draw an arrow that points to an object on the heap.

In this notation, *references* are always represented with arrows that start with a circular dot, which resembles and should remind the dot used in the source code to access fields or to call methods. For the special reference `null`, we draw a cut arrow that does not leave the small box.

The heap is drawn on the right and contains all the objects allocated on it, represented as rectangles. Each object is labelled with the name of its class and contains all the fields as smaller rectangles. The same convention of variables in the stack is adopted: fields are labelled with the name and their type, which is in turn treated appropriately.

Figure 2.1 shows a Stack and Heap diagram for the code written in Listing 1, assuming that the program is started by someone magically calling the static method 'C.run()' (like the main method).

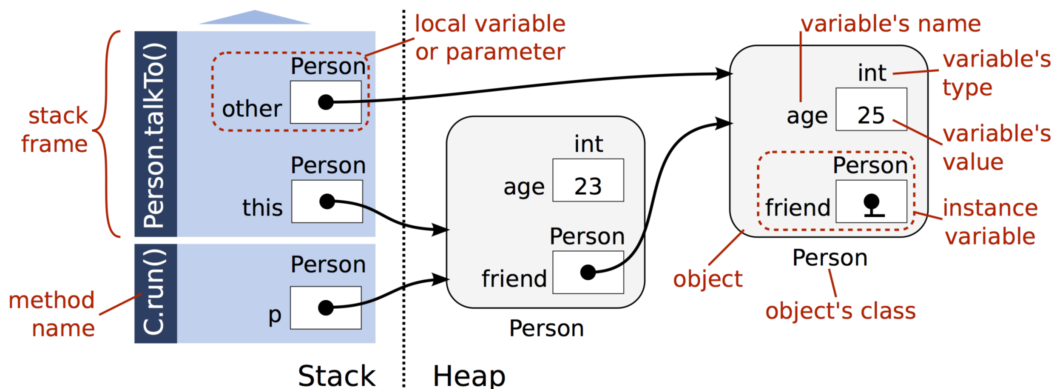


Figure 2.1. Stack and Heap diagram notation. Source: PF2 course material.

The notation for the heap is inspired by BlueJ, the IDE that accompanies the book used in the course [Barnes et al., 2006]. Our version, however, aims to fix some oddities of that representation, particularly that the stack is not shown and that objects are named, which proved to be a potential cause for misconceptions when learning about aliasing.

2.4.2 Sequence diagram

To understand interactions among objects and the different types of method calls we use a representation called *sequence diagram*. This diagram is usually formalised as one of the UML diagrams and it contains an enormous number of features. As we are in an introductory programming course and not in a software engineering one, here we limit ourselves to its most salient characteristics.

Objects are represented as rectangles with a line that goes downwards to match the direction of the time. Sticking to the UML notation, the rectangle contains a label of the form `name : Class`, even though as discussed in the previous section this might lead to problems when talking about aliasing, as objects do not have "names".

Method calls are represented as horizontal arrows that create new activation records, which are in turn shown as small rectangles on the lines of life of the objects. Returns are also represented as arrows in the inverse direction of the call; in fact, if we ig-

```java
public class C {
  public static void run() {
    Person p = new Person(23, new Person(25, null));
    p.talkTo(p.getFriend());
  }
}

public class Person {
  private int age;
  private Person friend;
  public Person(int age, Person friend) {
    this.age = age;
    this.friend = friend;
  }
  public Person getFriend() {
    return friend;
  }
  public void talkTo(Person other) {
    // Stack and Heap diagram at this point
  }
}
```

Listing 1. Example code for the Stack and Heap diagram.

nore exceptions, we can always say that for each call there is a matching return and, consequently, for each call arrow there must be a return arrow.

At any given moment, the method on the top of the stack which is currently being executed is highlighted. Calls to methods on the same object, regardless of being recursive or not, are placed one next to the other in a stacked fashion.

An annotated example of a sequence diagram for a sample code written in Listing 2 is shown in Figure 2.2. We assume that an instance `g` of the class `Game` already exists and someone has called `g.simulateStep()`.

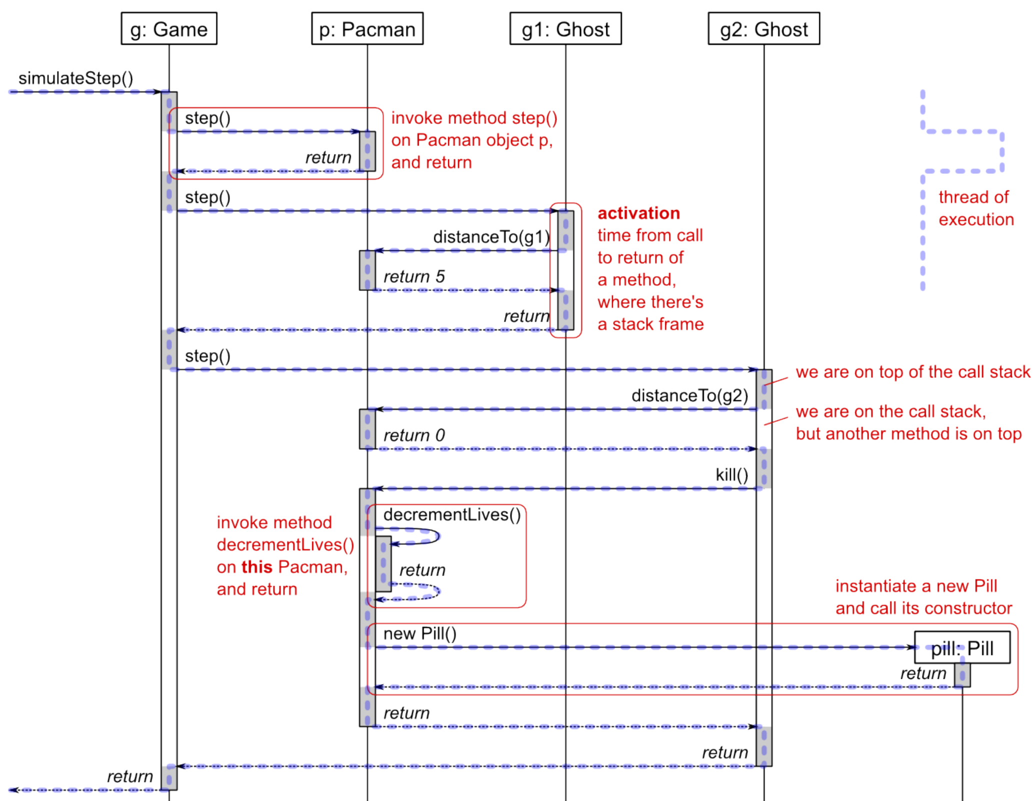


Figure 2.2. Sequence diagram notation. Source: PF2 course material.

2.4.3 Expression tree

Java code written in real-word projects often uses articulate expressions that, for example, call multiple methods on the same line using nesting or chaining. Breaking down complex expressions into small pieces is essential to understand how it is going to be evaluated and, among other things, what is the type of the whole expression. This takes advantage of the Java type system that allows determining and type-checking expressions at compile time.

```
public class Game {
  private Pacman p = new Pacman();
  private Ghost g1 = new Ghost(p);
  private Ghost g2 = new Ghost(p);
  public void simulateStep() {
    p.step();
    g1.step();
    g2.step();
  }
}
public class Pacman {
  public void step() {
  }
  public int distanceTo(Ghost g) {
    return ...;
  }
  public void kill() {
    decrementLives();
    new Pill();
  }
  public void decrementLives() {
  }
}
public class Ghost {
  private Pacman p;
  public Ghost(Pacman p) {
    this.p = p;
  }
  public void step() {
    if (p.distanceTo(this) == 0) {
      p.kill();
    }
  }
}
public class Pill {
}
```

Listing 2. Example code for the sequence diagram.

The expression tree breaks down an expression into atomic components such as literals, variables and single method calls. Each subexpression constitutes a node in the tree and has an associated type. The tree is evaluated from the leaves to the root which emits the final result of the evaluation of the whole expression. Nodes at the same level of depth are evaluated from left to right.

The expression tree that corresponds to the last line in Listing 3 is shown in picture 2.3.

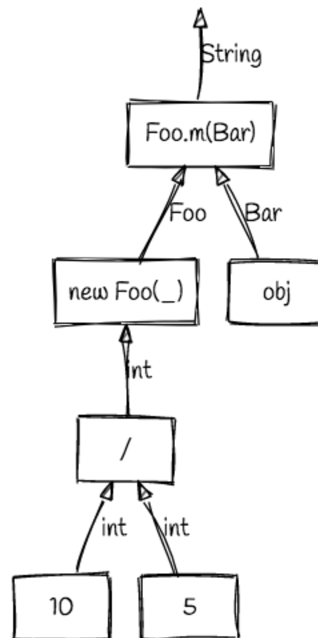


Figure 2.3. Expression tree notation.

2.4.4 Control-flow graph

The last notional machine introduced in the course, the control-flow graph (CFG), aims to improve the understanding of what happens *inside* a method: it is an intra-procedural analysis. Its main purpose is to explain the different programming constructors for selection (`if`, `if-else` and `switch` statements) and repetition (different fashions of `for` and `while` loops). It also expresses the semantics of short circuit operators and the conditional operator.

Formally, the graph starts from an entry node which has exactly one outgoing edge. The usual convention of representing statements with rectangle boxes and conditions as diamonds is followed. We also enforce the well-formedness of the graph requiring that the end node must be reachable from any other node of the graph.

An example of a CFG for the `foo()` method shown in Listing 4 is drawn in Figure

```
public class Foo {
  ...
  public Foo(int x) {
    ...
  }
  public String m(Bar bar) {
    ...
  }
}

public class Bar {
  ...
}

Bar bar = new Bar();
new Foo(10 / 5).m(bar); // expression tree for this line
```

Listing 3. Example code for the expression tree.

2.4. Note the special care dedicated to the short circuit operator && which effectively splits the condition.

```
public void foo() {
  bool c1 = true;
  bool c2 = false;
  if (c1 && c2) {
          m();
  }
  n();
}
```

Listing 4. Example code for the control-flow graph.

2.5 Programming misconceptions

The idea that while learning one develops *misconceptions* has been widely explored in pedagogy and has found confirmation in almost every domain. What is not obvious is how to *define* what a misconception is and how the term relates to similar concepts often explained using slightly different words. diSessa [diSessa et al., 2014] traces back the idea of misconceptions in educational research to the mid 1970s. They are described

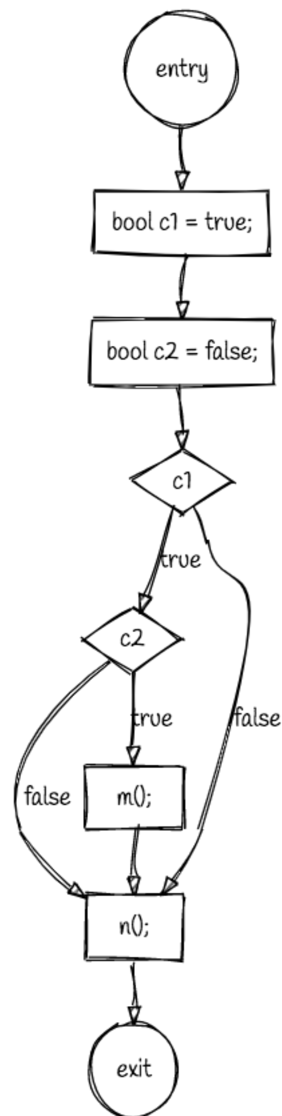


Figure 2.4. Control-flow graph notation.

in very different ways in the literature; some recurrent terms are "false beliefs" and "critical barriers". A common feeling is that students have those "conceptions" either because of previous knowledge (preconceptions) or because of poor teaching and poor understanding of the basic concepts in a field.

We can consider as an example the domain of physics. In this context, a misconception can be the belief that "an object's speed is proportional to the force on it". We know from Newton that this is not the case, as the force is proportionally related to the acceleration, not the velocity. Hundreds of instances of specific errors have been collected throughout the decades, although it is still disputed what constitutes a misconception and which are the boundaries between different ones. A big problem resides in the fact that misconceptions are often intertwined and developed in clusters, rather than in isolation.

The third section of Sorva's dissertation [Sorva et al., 2012] contains a great analysis of *programming misconceptions*. He acknowledges their pervasiveness in learning to program: "misconceptions of even the most fundamental programming concepts, which are trivial to experts, are commonplace among novices and challenging to overcome" [Sorva et al., 2012]. Many attempts have been made to try to categorize ways in which programming students struggle with core topics and exhibit all kinds of partial or even incorrect understanding of how things work.

In appendix A of [Sorva et al., 2012], 162 different misconceptions obtained from the author's experience and from the literature are listed. However, Sorva himself is sceptical about the intrinsic coherence of that list: "this is a list of not only apples and oranges, but also of tomatoes and the odd dried plum" and warns that the real commonality is just that those misconceptions are a description of real difficulties expressed by real students after an introductory course in programming.

There are different ways to *discover* misconceptions. One method comes directly from student descriptions of concepts, which in turn may be obtained in various contexts: interviews, think-alouds or drawing sketches of concept maps arguing about relations among concepts.

More original ways to elicit such misconceptions have also been explored. In [Hauswirth and Adamoli, 2017] authors describe a blended learning system, dubbed Informa, used during the course. Following the "active recall" technique, students are asked to summarise what they have learnt after each unit, effectively practising information retrieval. Studies shown that enforcing the habit of this practice increases the effectiveness of learning [Karpicke and Roediger, 2008]. These recall statements can also be analysed to discover a rich set of new misconceptions as, in theory, they contain words exclusively written by students. This approach, however, is subject to cheating in many forms: entered statements can be copied directly from book phrases, from a resource found on the internet or from colleagues.

2.6 Mastery Checks for research

In our previous experiences, we offered weekly mastery checks as a learning opportunity throughout the course. A teaching assistant sits together with a pair of students to assess their understanding of a given topic. We used to group more than one topic in a single session, choosing the ones with a strong interrelation, to save human resources, explore topics better and connect concepts.

We felt appropriate to use individual mastery checks also for doing research. They offer a unique environment in which to observe every aspect of a solution, including the approach students take to get there.

A significant part of the research body looks only at the artifacts that students produce in different forms: multiple-choice or free-text answers, drawings, and pieces of source code. However, we are also interested in capturing and understanding the *path* they take to come up with a solution. The process can reveal as much if not more information as the final result. One can consider whether the first response was immediate or if they waited a bit of time to reflect, how many corrections they made and how many times they changed their minds and started over, the degree of confidence demonstrated, the number of clarification questions asked, how seemingly unconnected aspects of different questions are related, and much more.

The constant interaction between the interviewer and the student furthermore allows recovering from cases where a student gets stuck and needs a little nudge to proceed. Proceeding differently would result in a blank sheet of paper that does not tell much about the real status of knowledge except that it is not perfect.

For these reasons, we designed a qualitative study centred on mastery checks as a mean to collect data. We explore all the aspects of its design in the next chapter.

Chapter 3

Designing the qualitative study

This chapter outlines the rationale behind the choices on how to accomplish our qualitative research study. In the next sections we will describe its various aspects, starting from the initial research questions that originated the study to all the details, technicalities included, on how it was conducted.

3.1 Microgenetic method

In science education, the *microgenetic* method is a popular way to conduct studies. The idea is to repeatedly observe a phenomenon in the same setting at different points in time to learn details on how and when learning exactly occurs. Chinn and Sherin provide multiple reasons to justify this method: "learning is not typically understood to be a rare and dramatic event", "mastery of significant knowledge often requires a very long period of time", "learning [...] occurs in parallel on multiple fronts" and "learning and learning events are heterogeneous" [Chinn and Sherin, 2014].

Microgenetic studies have been carried out also in the context of learning to program to understand debugging strategies [Lewis, 2012] and learning trajectories of youngsters (a 10-year-old girl, specifically) at their first attempt at programming [Pantic et al., 2016].

In this study we followed the microgenetic approach to track and understand students' progress in USI's Programming Fundamental 2 course, closely monitoring the status of their knowledge and their abilities with respect to the specific topics and skills they are expected to learn by the end of the course (see Section 2.3).

Students who participate in the study are examined in a series of weekly mastery checks. They are asked different kinds of questions which can be categorized into:

- definitional: define a concept or explain your understanding of a specific term;
- tracing: given a piece of code trace each step of its execution, possibly with the help of one of the four notional machines introduced in the course (see Section

2.4);

- coding: solve an algorithmic task writing Java code and use the appropriate constructs when explicitly asked to do so.

3.2 Research questions

This study has three main goals. Firstly, we want to investigate the *conceptual change* in understanding the syntax and the semantics of Java, that is when and how one acquires the concepts related to the programming language. Secondly, we want to expand and elicit a rich body of misconceptions that students commonly develop when struggling to grasp difficult concepts. Thirdly, we would like to see at a higher abstraction level the *strategies* used to solve programming tasks and analyse shifts in their use.

While specific misconceptions and higher-level strategies pertain to two different types of knowledge, we cannot study them in isolation as they are clearly interwoven and are exhibited concurrently in a mastery check session.

This study constitutes an initial attempt to answer the following research questions:

- What are the most common misconceptions developed during a programming course?
- How broad is, if any, the observed change about a misconception across multiple weeks for the same student? Do teaching interventions reflect on it?
- What strategies, either good or wrong, are internally developed by students to solve programming problems?

This initial qualitative study is also intended to serve as a fertile ground for further quantitative studies aimed at investigating hypotheses that will emerge from the analysis of the current one.

3.3 Ethical and privacy issues

Every research that involves humans has to pay uttermost attention to ethical and privacy issues.

This study records students and their interviewer during ten mastery checks. These sessions are video recorded with two cameras, one directed to the faces of the two subjects and the other that records a sheet of paper placed on the table from above. The screen of the laptop on which the coding takes place is recorded through a screen capturing tool.

Audio is being recorded by all these three sources and from a Google Pixel to obtain an automatic transcription of the interview. While there are many tools to do automatic

captioning of a video, we excluded all the cloud-based ones to keep sensitive recordings away from the Internet. We used, instead, the Recorder app[1], released by Google and available only on Pixel phones, that provides real-time *offline* transcriptions using on-device machine learning algorithms[2].

No recording is ever uploaded to the Internet. Video and audio files are stored and backed up on a NAS accessible only from USI's local network. Accounts to access the NAS are protected with passwords and limited only to the minimal requirements to complete the job.

It is not possible to anonymise recordings as videos contain students' faces and the audio contains their voice; however, the research output will not include any personally identifiable identifier (PII). Where applicable, results will be presented in aggregated form.

Students are remunerated with a voucher of 50 CHF for their participation and are given an informed consent form before their recruitment in the study. The consent form states what they are required to do, which information we collect about them and how it is treated. Students must give explicit permission about every single use of their data and they can withdraw from the study at any time.

3.4 Student recruitment

Edition 2020 of USI's Programming Fundamentals 2 targeted about 45 students. We were able to recruit six of them at the beginning of the course. All of the six participants are first-year students enrolled in the Bachelor program and are attending the course for the first time. All participants were male and therefore the (very limited) use of masculine adjectives and pronouns in the rest of the thesis does not leak sensitive information. We refer to the six participants with anonymous identifiers (P1 to P6).

Due to the special situation caused by the coronavirus pandemic, we were able to record all ten sessions with four of them, nine sessions with one student and four sessions with the remaining one.

3.5 Technical aspects

Mastery check sessions have been recorded using the following equipment with these settings.

- Two GoPro HERO 8 placed one in front of the student and the interviewer and the other, with the aid of a stand, roughly 60 centimeters above the table to record the handwriting on an A4 sheet of paper.

[1] `https://play.google.com/store/apps/details?id=com.google.android.apps.recorder`

[2] At the time of writing, it uses Tensorflow Lite.

Although this kind of cameras are able to record up to 4K videos at 60 frames per second, this configuration has at least two major blocking issues: the camera overheats after about ten minutes of recording and output videos are encoded with the new HEVC codec to keep the size reasonable (it is not possible to use the widespread MP4/H264 coded). For these reasons, we decided to film the scene with a 4K resolution at 30 frames per second.

Each GoPro also records the audio with a sampling frequency of 32 KHz.

- A laptop with macOS that uses the integrated screen recording tool to capture a 1440p video with a 44.1 KHz audio.
- A Google Pixel phone which records the audio at 44.1KHz and produces a textual file with the transcription.

Videos stored on GoPro cameras are split into chunks that do not exceed 4 GB, to allow the use of FAT32 as filesystem for the storage. Even when memory cards are formatted with newer filesystems that do not have the 32 bits limitation for sizes, videos are still split. To recombine them without the need of long transcoding, we used `FFmpeg`[3], the de-facto standard solution to convert audio and video streams.

A major challenge is to integrate all these sources to end up with a single edited video, with perfectly synchronised audio and transcription, and with proper cuts to select at any given moment the best of three sources: front camera, top camera or screen recording.

As this process can become incredibly long if done manually for dozens of videos to the point of reaching the complete infeasibility, I have created an automated Python script that does all the job with the help of several external pieces of software and libraries.

Prerequisites for the script consist of the three video files transcoded to 1080p resolution and the audio and the transcription files shared by the phone. If needed, HandBrake[4], an open-source video transcoder, can be used to easily reduce the resolution of the raw files.

The script takes advantage of the following external tools:

- `FFmpeg` to perform all the manipulations on both audio and video files.
- `sync-audio-tracks`[5], an open-source software that compares two audio tracks and tries to align them in the best way possible using cross correlation on their Fourier transforms. It produces the number of seconds (with millisecond precision) required to shift the second track to align it with the first one.

[3] https://ffmpeg.org
[4] https://handbrake.fr
[5] https://github.com/alopatindev/sync-audio-tracks

- `OpenCV`[6] to process frames from video files and compute differences among them.
- `NLTK`[7] to split Pixel's transcriptions into sentences.
- `aeneas`[8] to align an audio file and its textual transcription. This task is called "forced alignment": essentially we want to generate a synchronisation map between a list of fragments, which in our case are represented by split sentences and the audio file. One of the many supported output formats is the widely known SubRip subtitle file format (extension `.srt`) which can be embedded into `.mp4` container files and displayed by popular video players.

The lengthy process to produce the final edited video files consists of these eight steps:

1. Audio tracks sampled at 32 KHz frequency, with re-sampling where applicable, are extracted from all the video files and from the audio file using `FFmpeg`.
2. Offsets between the following pairs of audio files are determined using `sync-audio-tracks`: screen and front camera, screen and top camera, front camera and phone. The screen file is assumed to be the correct one, with the two cameras that start their recording earlier. Audio and video from these two sources are thus truncated, discarding earlier portions.
3. Using `OpenCV` all the frames from the screen recording video and from the top camera are read. The goal of this and the subsequent steps is to automatically determine interesting segments of these videos and prioritise them in the final version over the front camera which just shows participants' faces. The desired order is: screen first to show the coding part (ignoring involuntary movements of arms captured by the top camera), secondly the top camera to show drawings on the sheet and, as last resort when no movement is detected on the other sources, the front camera. Given this purpose, frames from these files are read only in grayscale to reduce at one third the memory occupation and with a resolution of 216p which is deemed appropriate for the job. Proceeding this way has also the benefit of speeding up significantly the calculation of the difference between two frames: reducing at one fifth each dimension compared to the original 1080p resolution leads to a reduction of 25 times in terms of the number of pixels contained in every frame.
4. Frames are processed using `OpenCV`'s `norm` function to compute the absolute difference, equivalent to the L1 norm, between one frame and its immediate successor. Frames are grouped in chunks corresponding to half a second; a score is

[6] https://opencv.org
[7] https://www.nltk.org
[8] https://github.com/readbeyond/aeneas

assigned to each chunk based on the sum of absolute values of the differences of the pairs of frames in it over the total number of pixels. A threshold value, empirically determined at 1/1000 for the screen and 1/10 for the top camera, is used to discriminate between moving chunks, those with a score higher than the threshold, and non-moving ones.

To compensate for spurious values that can occur for a variety of reasons, we "link" together two moving chunks with a non-moving one in the middle considering all of three as moving. At this point, isolated moving chunks, defined as such by not having moving chunks as neighbours, are considered as non-moving.

5. The final edited video is produced writing frames from the "most interesting" chunk considering videos from the screen recording, the top camera and the front camera in this order to account for priorities, as mentioned above. Only chunks marked as "moving" in the previous step are considered for inclusion in the edited video; in case of neither the chunk from the screen nor the one from the top camera are flagged as moving we resort to using the chunk from the front camera.

   In this step `OpenCV`'s `VideoCapture` and `VideoReader` classes are used to read frames from the original videos and to write the edited one. Note that even though computations were done on the lower-quality versions, at this stage we reconsider full-resolution videos to produce a high-quality result.

6. The audio from the front camera, which is supposed to be the clearest one as it faces participants' mouths, is added to the video using `FFmpeg` and included in a `.mp4` file.

7. Pixel's automatic transcription is split into sentences written one per line in a textual file. To accomplish this task `NLTK`'s sentence-tokenizer is used.

8. After prepending silence to the phone's audio to align it with the one recorded by the front camera and thus with the final video, `aeneas` is run to obtain aligned subtitles using the sentences from the previous point in the form of a `.srt` file. Subtitles are finally embedded in the final `.mp4` file using `FFmpeg` and optionally also stored as an external file (which might be convenient to work with software that does not recognise embedded subtitles as MAXQDA, see Chapter 5).

Due to the coronavirus pandemic, we were unable to continue doing the study in person after the first week of sessions. We shifted to an online version that basically dropped the two cameras while keeping only the screen recording and the transcription done on the phone. To accommodate these changes, an alternative version of the script has been developed; it basically ignores steps 3 and 4 of the previous list going directly to step 5 where it simply copies the screen recording video to the output, as there is no need of editing at all. The part of the script that deals with the transcription remains untouched.

While the script is not the primary intended outcome of this research project, we believe that it has great potential and many other research groups that regularly do qualitative studies, for instance in social sciences, may benefit from its use. Compared to the tedious manual work, this automated script makes research much quicker and potentially enables to record more sessions. For these reasons we intend to publish it as a side-contribution at an appropriate venue and release it as open-source software, also with the ultimate goal of reducing the technicalities required to know in order to use it.

Chapter 4

Mastery Check sessions

We carried out the qualitative study outlined in Chapter 3 during the 2020 edition of the Programming Fundamentals 2 course at USI (of which the contents and learning goals were presented in Section 2.3). In this chapter, we are going to describe each of the ten sessions of mastery checks which were held with the six participants. Each session lasted approximately 30 minutes, for a total of roughly 1600 minutes (more than 26 hours) of recordings.

Professor Hauswirth played the role of the interviewer during the first session, while I personally conducted the remaining nine sessions. I also was directly in charge of the recording and the technical aspects for all the ten sessions using the tools I developed (see Section 3.5).

The schedule for the sessions is motivated by the fact that we wanted students to acquire some knowledge about a certain topic before investigating their understanding of it. We kept a certain lag between the first time a new Java concept or a notional machine was introduced before assessing mastery about it during a session. As the sessions were recorded during multiple days during the week they are listed at, one can consider that only the topics introduced in the previous week and not those introduced in the same one were acquired to a certain degree by the students. The full schedule is presented in Table 4.1.

It is also important to highlight that we did not tell students in advance which topics they were checked on because we wanted this to be as close as possible to a "natural snapshot" of their mind. If participants were explicitly made aware of the topics, they would probably tend to concentrate their study time on those parts of the course material, leading to overfitting. Instead, we expected them to grasp the concepts simply by working through the ample learning material available on the course platform. The platform is called Informa and a brief overview of it is given in [Hauswirth and Adamoli, 2017]. We deemed that the book, the theoretical and practical study tasks accessible on Informa, and the "labs" in which students learn to code in Java in the context of guided programming projects already contained enough learning opportunities.

Some topics and, more commonly, notional machines recur multiple times in the study. We deliberately choose this in order to be able to perform analysis of the learning trajectories across the course (see Chapter 7).

Course week	Mastery check	New Java language topics
Week 1	/	Classes versus Objects, Method Invocation, Types, Method Implementation
Week 2	/	Variables
Week 3	Session 1	Literals, Operators, Expressions, References
Week 4	Session 2	Use of generics, Packages
Week 5	Session 3	Autoboxing, Static Members
Week 6	Session 4	Array Basics, Array
Week 7	Session 5	Enums
Week 8	Session 6	/
Week 10	Session 7	Inheritance
Week 11	/	Polymorphism
Week 12	Session 8	Abstract classes and Interfaces
Week 13	Session 9	/
Week 14	Session 10	Exceptions

Table 4.1. Mapping course weeks, mastery check sessions and Java topics.

The rest of the chapter is divided into ten sections, one per every session of mastery check with the summary of the main topics in scope that week, the questions posed, the "starter code" prepared beforehand that served as a starter for the discussion and was used as a template for further implementations, and the notional machine(s) used, if any.

4.1 Session 1: Classes versus Objects and Method implementation

The purpose of the first session was to elicit students' understanding of the difference between what a class is and what an object is. These terms are pervasive in object-oriented programming but they are abstract and may sound odd to novices.

We asked the students to explain their thoughts about the two concepts and a comparison with similar terms like "model" and "prototype". We found it surprising that most of them consider "model" a synonym of an object rather than a class, with justifications like "a (car) model is the concrete thing, not the sketch of the car you draw on

paper to design it".

Students had to complete the implementation of a simple `Calculator` class adding an instance variable of type `double` with name `value`, a parametrised constructor, the getter and the setter for the field, and two methods to add a number to the current value and to clear it. In this context, we asked about the use of `this` when having a parameter named the same as a field.

Then, we provided the following source code to use the class:

```
public class Demo {
  public void run() {
    Calculator c = new Calculator();
    c.setValue(3);
    c.add(5);
    int r = c.getValue();
  }
}
```

After explaining which methods *mutate* the object, we asked them to draw a Stack and Heap diagram for the `run` method. Figure 4.1 is an example of one of these drawings.

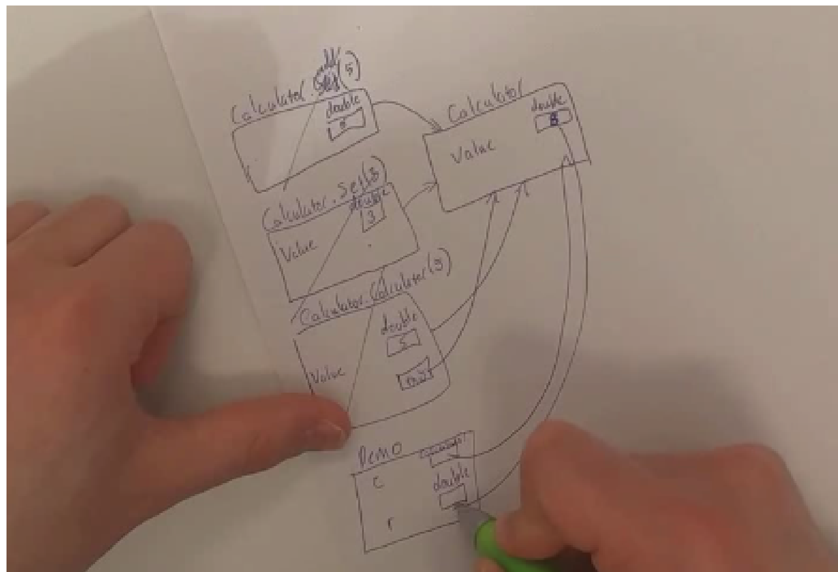


Figure 4.1. Stack and Heap diagram produced by a student during the first mastery check.

4.2 Session 2: References and Stack and Heap

References have been the main focus of the second session. References can be tackled from many different perspectives: at the two extremes, one can dive into the details of computer architectures and memory addresses or can stay at a very high level of abstraction, as we do in the Stack and Heap representation, using arrows as graphical elements to signify a memory reference.

This session assessed participants' competence about the "degree of sameness" of two Java objects. One can say that two objects are equal when they have the same *state*, such as two strings with the exact same characters, or when they are exactly the same object in the heap.

We considered a simple `IntHolder` class that contains just an integer field:

```
public class IntHolder {
  private int value;
  public IntHolder(int value) {
    this.value = value;
  }
  public void setValue(int value) {
    this.value = value;
  }
  public boolean isSame(IntHolder other) {
    // TODO: is this and other the same object?
  }
  public boolean isEqual(IntHolder other) {
    // TODO: do this and other have the same state?
    // Do we need to add a getter method for accessing other.value?
  }
}
```

We asked students to implement `isSame` and `isEqual` while paying attention to field access. More than one student said that we could not access the field of the `other` object, showing a misunderstanding about the meaning of the access modifier `private`.

We finally used the class in a `Demo.run()` method where we declare some local variables and instantiate objects.

```
public class Demo {
  public void run() {
    // TODO: initialize h0 so it does not refer to anything
    IntHolder h0
    IntHolder h1 = new IntHolder(5);
    IntHolder h2 = new IntHolder(5);
    IntHolder h3 = h2;
```

```
    IntHolder h4 = new IntHolder(8);

    // How many objects and local variables at this point?

    boolean b1 = h0.isSame(h1); // valid?
    boolean b2 = h0 == h1;
    boolean b3 = h1.isSame(h2);
    boolean b4 = h1.isEqual(h2);
  }
}
```

Common errors here regarded the `null` reference and the wrong understanding of it being a null object. This specific misconception could be caused by a lab students had to develop to experiment with different designs for list terminators, with one of them being the use of a "null object" to avoid `NullPointerExceptions`. This overloading of the term "null" probably confused lots of students and is a big warning sign for teachers.

Another error detected with the question after the five local variable declarations has been the interpretation of the assignment between two local variables of type `IntHolder` as a duplication of the object in the heap.

Most students were also not able to figure out that the call `h0.isSame(h1)` results in a runtime exception due to `h0` being `null`; it is however particularly interesting that some were able to detect this problem in the subsequent drawing of the Stack and Heap diagram.

4.3 Session 3: Method invocation and Sequence Diagrams

The third session was focused on *how* to invoke methods, which calls are valid or invalid and the interaction among objects, such as correctly identifying the caller and the callee in simple and complex expressions. This proved to be hard for some students in expressions which contained method chaining or nested method calls.

We gave them two classes: an `Engine` and a `Car` which has an engine as a field and manipulates it.

```
public class Engine {
  private int speed;
  public Engine() {
    speed = 0;
  }
  public void stop() {
    setSpeed(0);
  }
  public void setSpeed(int speed) {
```

```
    this.speed = speed;
  }
  public int getSpeed() {
    return speed;
  }
}
public class Car {
  private Engine engine;
  public Car() {
    engine = new Engine();
    engine.setSpeed(0);
  }
  public void speedUp() {
    engine.setSpeed(engine.getSpeed() + 1);
  }
  public void stop() {
    engine.stop();
    setBreaks(true);
  }
  public void setBreaks(boolean b) {
    // TODO
  }
}
```

Students had to complete the `Car` class adding a `boolean` field to represent the brakes, to call `setBrakes()` in the constructor to explicitly initialise the instance variable and to complete the implementation of the method.

After that, the rest of the session required a second notional machine, the sequence diagram (see Section 2.4.2 for a refresh on our simplified notation for it, based on the full UML version). Students drew the execution of the following four statements to demonstrate understanding about the order of the calls, the time at which objects are born, constructor calls and further interactions through internal and external method calls.

```
Engine e1 = new Engine().setSpeed(2);
new Engine().setSpeed(e1.getSpeed());
Car c = new Car().speedUp();
c.stop();
```

4.4 Session 4: Control Flow and Conditional Computation

The fourth session is the first one that includes a way to change the behaviour of a program depending on data: this is called *conditional computation*. In Java we have `if`, `if-else` and `switch` statements; all of them test one or multiple conditions to select the appropriate branch. To exploit them, we also consider necessary the knowledge of operators to combine multiple expressions into a compound one.

We also assume basic familiarity with the loop structure. This session required the implementation of two methods: one to print a classic countdown and one to sum all the elements of a list. Note that at this point of the course we have already talked about generics (see Table 4.1).

```
public void countDown(int n) {
  // TODO print all numbers from n, n-1, ..., to 0
}

public int sumList(ArrayList<Integer> list) {
  // TODO add the values of all elements in the given list
}
```

The third and the fourth methods, already implemented, were meant to assess students' ability to handle nested conditional statements such as an `if` inside a `for` loop and to check their knowledge about the semantics of the short-circuit and operator. We asked them to draw the corresponding Control Flow Graphs (see Section 2.4.4), the third notional machine under scrutiny. Figure 4.2 shows an example of such a graph drawn by a participant.

```
public int index(int[] values, int value) {
  for (int i = 0; i < values.length; i++) {
    if (values[i] == value) {
      return i;
    }
  }
  return -1;
}

public char firstChar(String s) {
  if (s != null && s.length() > 0) {
    return s.charAt(0);
  } else {
    return ' ';
  }
}
```

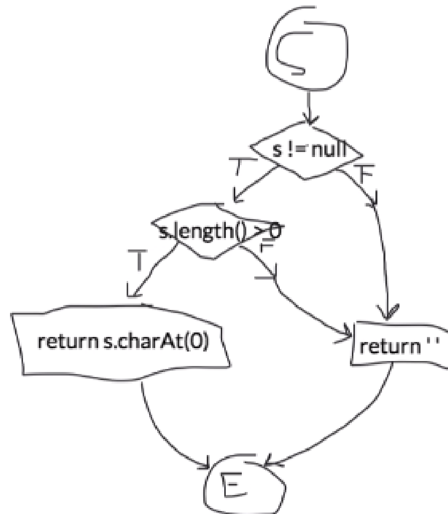


Figure 4.2. Control Flow Graph for the method `firstChar` produced by a student during the fourth mastery check.

4.5 Session 5: Recursive Computation and Iterative Computation

The fifth session used the only additional topic in terms of the Java language that was introduced, arrays, and was more directed towards the algorithmic and strategical thinking: how to solve a given problem in different styles. In particular, we were interested in observing programming competence with respect to recursion and iteration.

The relatively simple task required to find the value of the minimum element in an array in three different styles: recursively, with an "old-style" `for` loop using a counter and with the "new-style" `for-each` loop. The caller method looked like this:

```
public static void main() {
  int[] values = new int[] {15, -1, 2, 9};
  System.out.println(findMinRec(values, 0));
  System.out.println(findMinIt(values));
  System.out.println(findMinIt2(values));
}
```

Students have been asked to draw the sequence diagram of the recursive call; two of them could not produce a correct version of it showing symptoms that the formalism was not fully understood for "corner" cases, like recursion on the same object. In fact, during the class time a sequence diagram for a recursive method was presented, but this remarks the differences between structural recursion, like traversing a linked list, and generative or computational recursion, such as computing the factorial of a number.

An example of a wrong diagram that misses rectangles for activation boxes and consequently uses improperly the arrows for calls and returns is shown in Figure 4.3.

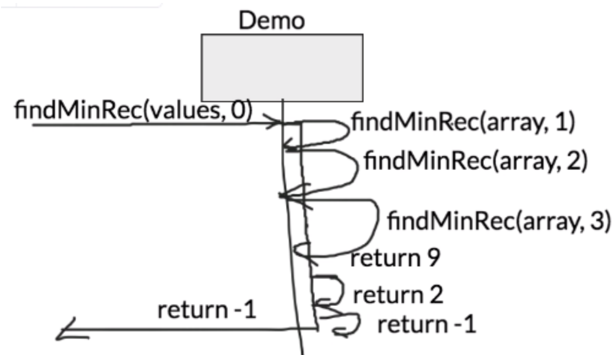


Figure 4.3. Sequence diagram produced by a student during the fifth mastery check.

4.6 Session 6: Recursive data structures and Variables

The sixth session was the first one after the Easter break. We felt that it was too early to put inheritance in scope as students had little practice with it, having not completed yet the relevant lab. Thus, we opted for a more in-depth assessment of recursion, focusing this time on structural recursion. We also talked about variables and the modifier `final`, which allows discussing immutability.

The starter code was about defining a class `Node` to be later able to build a linked list made of nodes and to perform basic computations with it, such as summing all the elements and retrieving the index of one of them.

```
public class Node {
  private Node node;
  private int value; // what about final?
  public Node(...
    ...
  }
  public int sum() {
    ...
  }
  public int indexOf(
    ...
  }
}
```

Perhaps the most difficult challenge was to complete the following statement to build with a single expression the whole list in the correct order.

```
// Construct a linked list [1, 2, 3]
Node head = ...;
```

The desired solution was to nest `new` calls and to use a constructor which takes a reference to a `Node` and prepends to it the new one, like this:

```
Node head = new Node(1, new Node(2, new Node(3, null)));
```

However, the above code requires significant experience and novices have seen it so few times that it is hard for them to process the numerous concepts contained in that expression: the `null` reference, constructor calls, method nesting, and the order of the list which furthermore depends on the particular way you chose to implement the constructor.

4.7 Session 7: Literals, Types and Expressions

The seventh session was all centred on expressions and thus included literals, their constituent pieces, and types, thanks to the strictness of the Java typing system. We tested the fourth and last notional machine introduced in the course, the expression tree (see Section 2.4.3) to explain thoroughly the order of evaluation and the types of each subexpression. In this context, we also checked the basics of implicit and explicit type casting.

We presented the following class `Circle` that has an instance field stored as `double`:

```
public class Circle {
  private static final double PI = 3.14;
  private double radius;

  public Circle(final double radius) {
    this.radius = radius;
  }

  public double getCircumference() {
    ...
  }

  public void copyRadiusFromCircle(final Circle circle) {
    this.radius = circle.radius;
  }

  public void makeDouble() {
    this.radius *= 2;
```

```
  }

  public String toString() {
    // "Circle of radius xxx"
    ...
  }

}
```

We then asked students to evaluate the following six statements which contain expressions with all kinds of characteristics: method chaining, method nesting, literals, implicit and explicit casting.

```
new Circle(10).getCircumference();
double halfCirc = 1/2 * new Circle(10).getCircumference();
new Circle(10).makeDouble().getCircumference();   // what is the type?
float radius = 2.5f;   // what if just 2.5?
Circle c = new Circle(radius);
c.copyRadiusFromCircle(new Circle(10 / 5));
```

We asked participants to draw the expression trees corresponding to the right-hand side of the assignment in the second statement and to the last statement. As an example, a tree drawn by one of them is shown in Figure 4.4.

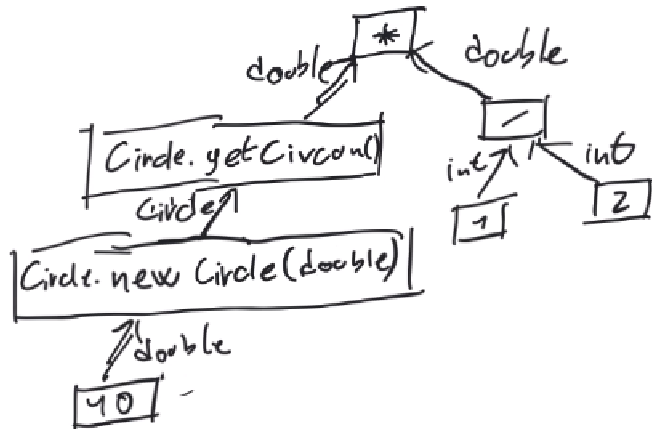


Figure 4.4. Expression tree produced by a student during the seventh mastery check.

4.8 Session 8: Inheritance and Polymorphism

The eighth session was the first one to target inheritance, one of the essential characteristics of object-oriented programming. The starter code presented two classes, one of

which inherits fields and methods from the other one. We explored calls to superclass constructors and methods, dynamic dispatch and type compatibility.

```
public class Employee {
  private int dailySalary;
  public Employee(final int dailySalary) {
    this.dailySalary = dailySalary;
  }
  public double getHourlySalary() {
    // 8 hours in a (work)day
    ...
  }
  public int getYearlySalary() {
    // 200 days in a (work)year
    return 200 * dailySalary;
  }
}

public class ProjectManager {
  private int bonus;
  public ProjectManager(final int dailySalary, final int bonus) {
    ...
  }
  public int getBonus() {
    ...
  }
  public int getYearlySalary() {
    // base yearly salary + bonus
    ...
  }
}
```

Classes were used by the following driver method.

```
public class Demo {
  public static void run() {
    Employee bob = new Employee(80);
    bob.getBonus();  // ok?
    ProjectManager alice = new Employee(100);  // ok?
    Employee alice = new ProjectManager(100);  // ok?
    alice.getHourlySalary() // result?
    bob.getYearlySalary()   // result?
```

```
    alice.getYearlySalary() // result?
  }
}
```

Students were also asked to draw a Stack and Heap diagram for the method. We uncovered new misconceptions related to the representations of inherited objects in the heap and calls to superclass methods and constructors.

4.9 Session 9: Use of generics, ArrayList versus array

The ninth session was a detour from the inheritance journey and was aimed to understand the use of generics and to compare arrays and `ArrayLists`. We briefly touched those topics in sessions 4 and 5, but at this stage we are approaching the end of the course and we saw an opportunity to dig a bit deeper.

The idea is to have a class which holds a list of contacts. To keep the implementation as simple as possible, each contact was only represented by a name and a phone number. Participants had to implement two versions of the class, one with two parallel plain Java arrays and the other using `java.util.ArrayList`. In both cases, requirements prescribed a method to add a new name to the list and to print all the names longer than six characters. Note that the former operation requires growing the array, creating a slightly longer copy of the original one. While this is not trivial for beginners, students already practised the same operation in an earlier lab.

```
import java.util.ArrayList;
public class ContactsAL {
  private ArrayList<String> names;
  public void addName(final String name) {
    ...
  }
  public void print() {
    ...
  }
}

public class Contacts {
  private String[] names;
  private String[] numbers;
  public void addName(final String name) {
    ...
  }
  public void print() {
    ...
```

```
  }
}
```

We also discussed whether we needed constructors or not, which types are allowed between angular brackets for generic types, which is the most appropriate type for storing phone numbers and how it is possible that each object has a `toString()` method. As usual, we employed the Stack and Heap representation to show the exact arrangement of frames and objects in memory when executing this simple method:

```java
public class Demo {
  public static void run() {
    new Contacts().addName("Luca");
  }
}
```

A snapshot of one of the constructed diagrams for the array version just after calling `addName` is shown in Figure 4.5.

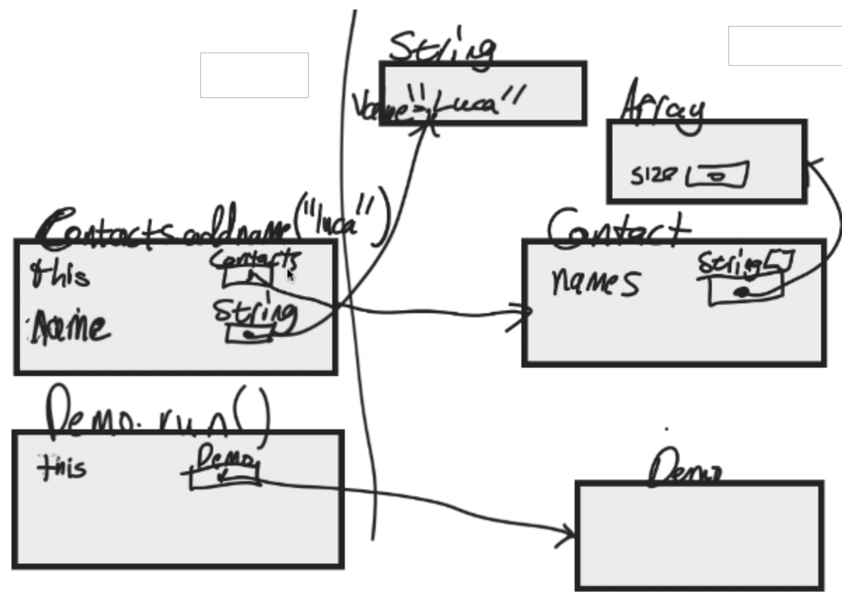


Figure 4.5. Stack and Heap diagram produced by a student during the ninth mastery check.

4.10 Session 10: Abstract classes and Interfaces

The tenth and last session was focused on the more advanced aspects related to inheritance, namely abstract classes and interfaces. We wanted to understand whether

students understand the new “constrains” one has to obey when inheriting from an abstract class or implementing an interface. Tangentially, we also wanted to check how participants dealt with more complex designs (three classes in this case).

```
public abstract class Animal {
  private String name;
  public Animal(final String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public abstract String getColor();
}

public abstract class Pet extends Animal {
  private String nickname;
  public Pet(final String nickname, final String name) {
    ...
  }
}

public class Dog extends Pet implements Speakable {
  public void bark() {
    System.out.println("Bau");
  }
}

public interface Speakable {
  public void speak();
}
```

Questions targeted the implementation of `Dog`, such as the need of implementing a constructor, the `getColor()` method due to inheritance and the `speak()` due to the interface. We also quickly revised what constitutes an expression using the following driving code.

```
public class Demo {
  public static void run() {
    final String name = "Dog";
    final String color = new Dog(name, "My" + name, "Brown").getColor();
  }
}
```

We finally asked them to draw the sequence diagram for the right-hand side of the last assignment and the Stack and Heap diagram for the `run()` method.

Chapter 5

Coding in MAXQDA

The whole study has been conducted with MAXQDA[1], a software for mixed-methods research. We imported all the edited videos of the sessions, along with the automatic transcripts obtained as explained in the previous chapter, into the MAXQDA document system.

MAXQDA offers the possibility to code, which means to "tag", segments of documents in any form. When your document is a piece of text, you can highlight words and sentences and code them. When your document is instead a video, you can select a chunk of it and code just that segment. Video files and their codes are shown in a timeline very similar to what one can find in video-editing software.

An overview of how all of this is shown in MAXQDA is offered in Figure 5.1. The left panel is split in two: the top part contains the documents, the bottom one lists all the codes. Both documents and codes can be grouped into a hierarchical structure (but just one-level deep). On the right, the timeline of a video is shown with the coded segments highlighted using coloured stripes.

5.1 A-priori versus open coding

In general, when coding documents one can follow two radically different approaches: a-priori coding or open coding. The former requires to prepare a list with all codes strictly before starting the actual process of coding; you are not allowed to change or add further codes during the process. The latter is way more flexible: even if you start already with a set of codes, you can invent and add new ones when you deem appropriate to do so during the work. Working this way allows the coder to make the best out of unexpected situations that arise all the time during the interviews and does not limit his or her creativity to come up with better codes.

[1] https://www.maxqda.com/

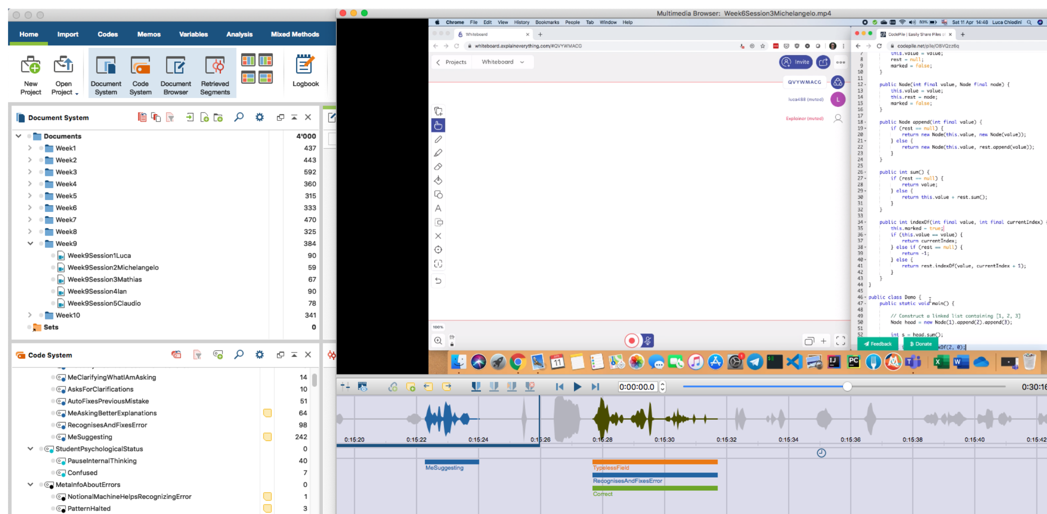


Figure 5.1. A screenshot that shows how MAXQDA looks like.

5.1.1 K-Java rules as codes

Our initial idea was to do a-priori coding borrowing the rules from K-Java, a complete formal specification of the Java semantics in the $\mathbb{K}$ framework [Bogdanas and Roşu, 2015]. $\mathbb{K}$ is a framework that allows specifying the syntax of a language as annotated Backus-Naur Form (BNF) and its semantics as reduction rules, repeatedly applied, over configurations. $\mathbb{K}$ rules describe how to transition from a configuration to another, until you end with a base one. The framework is designed to allow expressing language features as a small set of rules, keeping things concise.

K-Java defines 1074 rules to completely express the whole syntax and semantics of Java version 1.4. To give a sense of what a $\mathbb{K}$ rule looks like, Figure 5.2 shows the transformation of a `try/catch` block into a sequence of statements S to be executed, followed by catch clauses.

RULE TRY

$$\frac{\texttt{try}\ S\ \ \mathit{CatchList}}{S \curvearrowright \texttt{catchBlocks}\ (\mathit{CatchList})}$$

Figure 5.2. Rule for `try` in K-Java.

Similar rules exist for every single feature listed in the official Java Language Specification (JLS) of Java 1.4. A possible approach is therefore to treat each rule as a code and tag every usage of a Java feature during the interview with the corresponding code. Unfortunately, this methodology has some limitations:

- while the course does not cover advanced and more recently introduced Java fea-

tures (such as lambdas), the target is version 8 as we require the use of generics, which were introduced in Java 1.5 and are thus not available in K-Java;

- during the reduction process across configurations, 𝕂 loses all information about the original source code and thus is not able to map back the result to the original line of the source code;
- each Java language feature needs to be linked to many different rules, which are essential for the 𝕂 framework but completely inessential at this level of abstraction;
- the process would ideally take place automatically, but this would require firstly to being able to perform reliable optical character recognition from a screen recording (the current state of the art gives poor results on videos) and to inject codes into a MAXQDA project (this can be done as it is stored as a pure SQLite database, but the schema is undocumented and MAXQDA does not specifically support programmatic manipulations);
- significant parts of our interviews are not about the code per se but involve higher-order reasoning about the code, the notional machines or the abstract concepts which cannot be captured;
- K-Java is not working with the newer releases of the 𝕂 framework nor with the older ones and the original authors are not available to provide assistance to solve blocking issues.

5.1.2 Parsing to discover codes

Another more lightweight approach is to use a simple parser to traverse students-produced code and tag the relevant pieces. Open-source parsers, such as `javaparser`[2], compatible with the latest version of Java are available and have been evaluated. However:

- a parser ignores the semantics of Java features and misconceptions often are more related to how things work than the exact Java semantics;
- students often write incomplete code due to the nature of the question or are provided with code that on purpose does not compile, this might be alleviated with “fault-tolerant” parsers that attempt the job even on invalid code;
- all the issues described earlier related to the automatism of the approach, particularly the ones related to the automatic recognition of code from a video, still persist.

[2] https://javaparser.org/

All this considered, we opted not to go down the way of a-priori coding. We used human intelligence to directly analyse each recorded session to include every possible aspect: syntactic and semantic features of the Java language, higher-order strategies employed to solve problems, meta information about the structure of the session and general observations about the status of the student.

5.2 Codes

All the codes created during the process have been classified and divided into nine macro categories, represented as top-level codes. While MAXQDA considers the top-level codes as codes on their own, in our approach we are never going to use them directly, with the underlying assumption that when a segment is coded with one of the sub codes, it is also appropriate to consider that segment tagged with the super one (exactly as super and sub typing in class hierarchies).

The nine elements in the bullet list will provide information on the meaning of a certain category, as well as the list of all codes that are part of it with explanations on what they intend to represent.

- **Misconceptions** (63 codes)

  The category misconceptions contains one code per every misconception that was already identified by previous work in our research group [Hauswirth and Adamoli, 2017].

  Roughly 200 misconceptions were already collected and attributed to topics, allowing one misconception to be assigned to multiple topics. However, not all of them have been included in MAXQDA: we included only those for which we found evidence and that were addressed in at least one of our ten sessions.

- **NewlyDetectedMisconceptions** (127 codes)

  This category broadens the previous one adding to the already known misconceptions a set of more than 120 new ones. All these new misconceptions have been elicited through the careful analysis of the recorded sessions.

  Many new misconceptions target specific aspects of notional machines, characterising errors that students frequently make while trying to come up with their own version of them. This aspect is investigated in Section 6.4.

  Some others are more related to the syntax and the semantics of the Java programming language. We do not show the full list here as each misconception pertains to such a small aspect that it would not make sense to just go through them one by one. On the contrary, we try to organise them into meaningful structures and present a curated and hopefully interesting selection in the next chapter.

- **Correctness** (3 codes)

  Each segment of a video related to a misconception was tagged with the code of that particular misconception, taken from one of the two categories above, and a code that classified whether the student provided a *correct* answer or a *wrong* answer. In the former case, we claim that the student does not have that misconception, while in the latter we can hypothesise that the misconception is present.

  While we strived to divide only into correct and incorrect, in a very small fraction of cases (less than 0.5%) we used the code Maybe to mean that it was really impossible to understand whether the student had or not the misconception looking only at that segment and its context in that session.

- **Terminology** (2 codes)

  This category contains just two codes used to assess the mastery of the appropriate terms used in the context of object-oriented programming. The codes are ProperTerminology and ImproperTerminology.

  We deem this important as we often see students who struggle with dealing with the "language" of programming languages, that is the bag of terms which are commonly used by experienced programmers to clearly mean a specific thing. Not mastering these words can mean a subtle exclusion from a world where everybody speaks "the same language", either in programming websites or in an office with colleagues.

- **MetaInfoAboutQuestionsAndAnswers** (10 codes)

  This category contains codes to classify the questions asked by the interviewer and some meta aspects of the answers provided by the student. It is important to highlight that codes are not exclusive.

  We have four codes related to the questions:

  - AskingQuestionSecondTime, when the interviewer had to ask a question twice to get a proper answer;
  - ClarifyingWhatIAmAsking, when the interviewer clarified the question asked in precedence, possibly because the student asked for a clearer question or because he seemed to have misunderstood the original one;
  - AskingBetterExplanations, when the interviewer wanted to dig deeper into a student's response, maybe because the student was not completely answering the original question;
  - Suggesting, when the interviewer had to explicitly suggest a partial or a full answer to the student: often this codes a hint given when the participant got stuck;

four codes related to the answers:

- SaysUncertaintyExplicitly, when students said aloud that they were unsure on how to proceed or solve a given task;
- DoesNotKnowConcept, when students clearly do not know a concept or a term's meaning at all and thus we could not explore that aspect in the session;
- ProbablyMisunderstoodWhatWasAsked, when students show symptoms that they have not correctly understood the question and started answering an unrelated one;
- AsksForClarifications, when students recognise to have not understood well the question and want a better explanation;

and finally two codes that assess the student's behaviour after making a mistake:

- AutoFixesPreviousMistake, when students *without* external interventions or suggestions recognise an error they made earlier and correct it;
- RecognisesAndFixesError, when students, after being told to recheck a piece of the work they did, recognise the error and fix it.

- **StudentPsychologicalState** (2 codes)

  This category is related to the psychological state of a student that is currently undergoing the mastery check. For this limited study, we tracked only two behaviours:

  - PausedInternalThinking, when students wait a significant amount of time, up to several seconds, before starting to answer a question: this shows that they are currently thinking about possible multiple strategies when confronted with an algorithmic task or that they are evaluating different alternative explanations for a concept or maybe even something else;
  - Confused, when students show to be clearly confused while answering a question, possibly contradicting themselves multiple times.

  We did not explore further these codes as they are a bit out of our area of expertise, but we nonetheless believe that this category combined with the previous one has great potential for learning and behavioural sciences to analyse these kinds of "reactions" or statuses of the different stages of a mastery check session, which is in itself a delicate moment since you have two people, one instructor and one student, sitting together in an exam-like situation.

- **MetaInformationAboutErrors** (15 codes)

This category shifts the focus from specific misconceptions, such as improper use of notional machines or a wrong belief about how a Java construct works, towards meta-information about those errors and their influence on the solving process. These observations are enabled by the ability to observe students *while* they solve a task, rather than just taking a snapshot at the end. They aim to provide, to the extent that is possible, reasons for why students did a mistake and elicit common patterns that may help to diagnose problems.

We have three codes related to notional machines which are discussed in Section 6.4:

- NotionalMachineHelpsRecognizingError, when a notional machine helped students to see a flaw in their solution they would not probably have caught in another way;
- DerivesWrongInformationFromNotionalMachine, when a notional machine, regardless of whether it contained errors or not, is suspected to misinform the student and cause errors;
- UncertainAboutIrrelevantFormalismOfNotionalMachine, when students direct their attention on how to deal with a specific aspect of a notional machine which is inessential and possibly was not fully formalised or clearly explained in the course;

and other more general codes:

- PatternHalted, when an error is made on a specific task that has a difference with those solved before it, which probably induced a pattern that no longer holds;
- RecurrentMistake, when within the same session a student repeats the same type of error, corroborating the hypothesis that the misconception is really present;
- WishfulThinking, when one devises the semantic of something (as Java operators' precedence) based on what it is more helpful to immediately achieve the goal, the wished behaviour, rather than on how it actually works;
- Inconsistency, when in the same session we observe two (or more) answers by the same student that are in conflict and that cannot possibly be all true at the same time;
- ContextNotSufficientlyAnalysed, when students provide answers without looking at the bigger picture: for example, when coding a solution they only look at the specific method they are implementing without analysing other methods and other classes;

- ChangesCorrectIntoIncorrect, when a student solves a given problem in the correct way at first, but later on revises the answer turning something that was correct in the first place into something else that is no longer suited;
- NeverExperiencedBefore, when students say aloud or demonstrate that this is the first time they approach a certain topic, hear a term or see a specific program construct;
- FlowOfThinkingLostAfterInterruption, when students are on a good track but their flow of thinking gets interrupted by the interviewer who asked a side question or by themselves to look, review or fix something else; this "distraction" is then not promptly abandoned and impedes the correct prosecution of the current activity;
- ErrorByWrongAnalogy, when a mistake is made by a wrong inferred analogy with a topic that has some similarities with the one students are confronted with; examples are extensively discussed in Section 6.3.
- WrongGeneralRuleInferredFromWrongExample, when an example that contains a mistake is instead believed to be correct and used to infer a general rule, which is then obviously wrong.
- DealsWithConsequenciesOfAnError, when students make a mistake that remains unfixed and it influences all the rest of the session, as they are not able to do certain things because they do not connect with the existing state of the notional machine or the source code;
- DoesNotQuestionTheAuthority, when students take for granted that the code they are provided with is always correct and does not need to be adapted as the rest of the program evolves.

• **ProgrammingPatterns**

This category contains codes related to the "style" of programming, both with respect to the specifics of the Java syntax and to general patterns common across languages and paradigms. Examples of the first type include the use of unnecessary parentheses in expressions or the use of this long snippet of code

```
if (cond) {
  return true;
} else {
  return false;
}
```

instead of just returning the condition. On the other side we can found generic examples of improper code conventions or choosing an inappropriate data type for a piece of information, such as integers to store phone numbers.

- **StrategyErrors**

  This category contains interesting codes we have created to describe something different from the very specific and limited in scope errors classified as misconceptions. Instead, taking advantage of the richness of information we have captured, we wanted to characterise broader programming behaviours. These codes could be viewed as "higher-level misconceptions", as they focus more on the *strategy* one needs to employ to solve a given problem rather than on specific programming constructs.

  We do not present here the specific codes but we refer to the discussions contained in the next chapter. In particular, there are many strategies related to the recursion (Section 6.5.1) and one interesting discussion on how students draw the Stack and Heap diagram (Section 6.4.1).

Overall, I assigned 4007 codes to video segments to enable the insights presented in the next chapters. The structure for the codes presented above is the result of refinements after multiple iterations.

As a final note, we want to remark that this classification is far from being "the correct one": one may come up with many additional criteria to classify recordings based on what is more interesting for a specific research. Also, it may be argued that some codes fit more appropriately into a different category: it is again a matter of subjective choices. We believe though that this is a good starting point for a taxonomy and that it can be improved in subsequent works.

Chapter 6

Insights about misconceptions and strategies for solving problems

In this chapter we are finally able to describe new misconceptions and interesting strategies for solving programming problems that were uncovered by the qualitative study. Codes related to these two categories have been introduced as NewlyDetectedMisconceptions and StrategyErrors in Section 5.2.

We felt it would have been pointless to provide a list of hundreds of them without any kind of grouping. Firstly, the scope of this study has been since the beginning broader than collecting a bunch of misconceptions; secondly, other lists of misconceptions, either short or long, already abound in the literature (just as an example, an extensive list can be found in [Sorva et al., 2012]); lastly, it is not very useful to provide them in an unstructured format.

Instead, we chose to present a selection of them, grouped into themes which are outlined by the titles of the sections of this and the next chapter. In each of the sections we provide justifications for the inclusion.

Misconceptions will be individually presented in a grey box that has the following form.

NAMEOFMISCONCEPTION	
Title	NameOfMisconception
Context	Prerequisites to understand the description of the misconception
Description	Textual description of the wrong thinking shown by the students
JLS	References to chapters of the Java Language Specification[a]
Observations	Sessions involved with this misconception

For most of them, in this space we provide one or multiple examples of specific instances of this misconception made by students, with excerpts from the source code they wrote and from the diagrams they drew.

[a]The Java Language Specification is a document that *fully specifies* the syntax and the semantics of a given version of Java. In this work we refer to Java 8; JLS for it can be found at `https://docs.oracle.com/javase/specs/jls/se8/html/index.html`.

In the next chapter we adopt the same format also for codes that are not really misconceptions, but target higher-level competencies such as solving strategies (see Section 5.2).

6.1 Several misconceptions are related to the programming language

As one might expect, the vast majority of the uncovered misconceptions are related to the syntax and the semantics of the programming language used (in our case, Java). We attempted to show all of them in Figure 6.1, even though the classification into syntax and semantics is very blurred. In fact, almost every aspect is tightly entwined with both characteristics such that it is almost impossible to consider them in isolation.

Some of them target very specific aspects of the language, such as WrongFinalPosition about the order of the keyword `final` or the self-explanatory DecimalLiteralsFloatByDefault. Others, instead, show deeper issues, such as TypelessField when students do not write the type of a field.

Having said that, it is questionable what constitutes a “small” error which scope is limited, can be easily fixed and does not prevent or interfere with learning versus a “bigger” error which has huge implications and shows a severe misunderstanding of the conceptual model. Sticking to the examples mentioned above, the seemingly innocuous DecimalLiteralsFloatByDefault may in future expand to a complete confusion about the Java type system, while on the contrary the more worrying TypelessField might just be an oversight that does not endure.

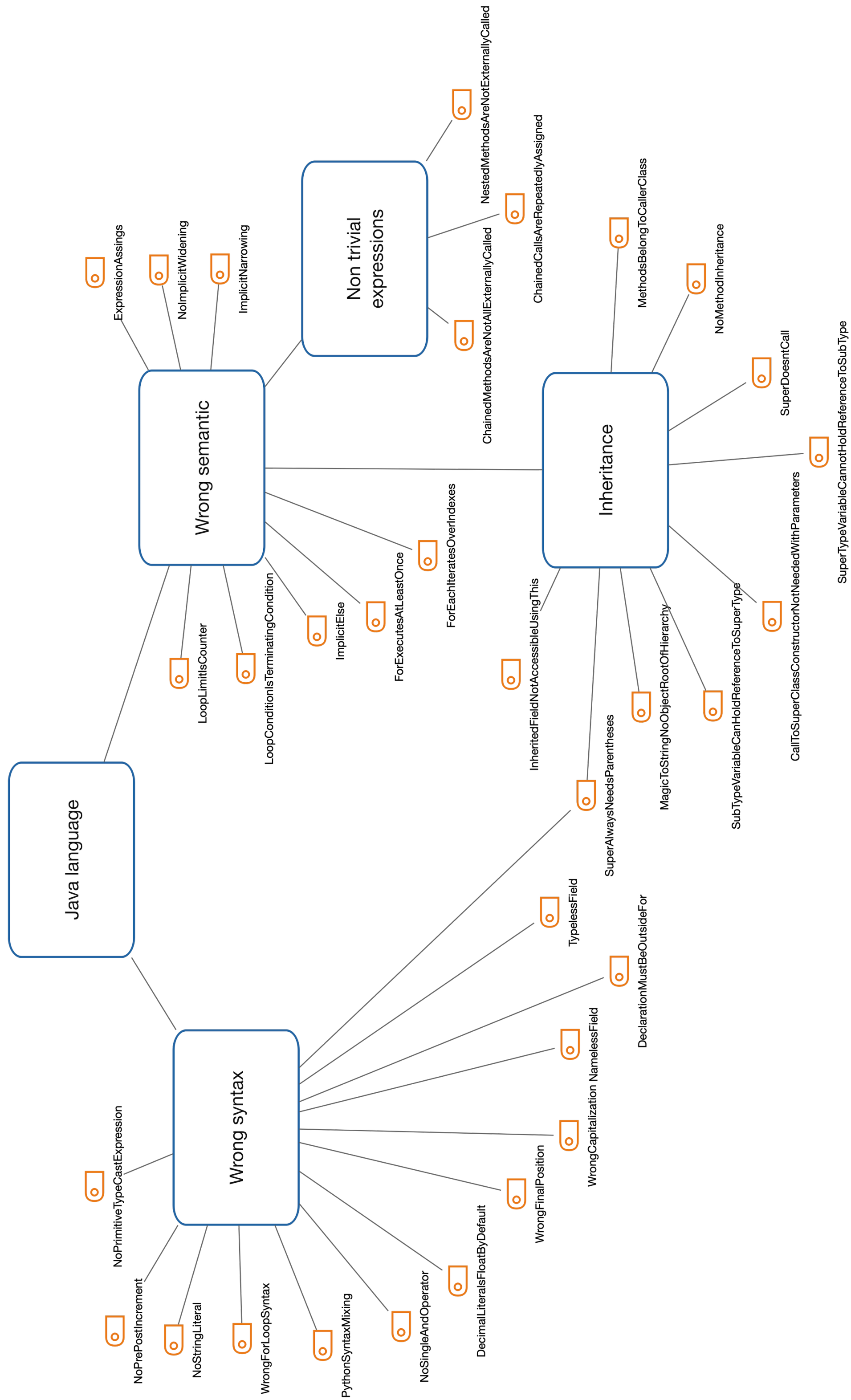


Figure 6.1. Map of codes about misconceptions related to the Java language.

6.2 Some misconceptions are one the dual of the other

The analysis of misconceptions has shown that a form of duality can be found in some of them. The word *dual* in mathematics refers to the interchange of particular pairs of terms to go back and forth from one concept to the other.

Consider the map shown in Figure 6.2 which shows three pair of related misconceptions which can be considered one the dual of the other.

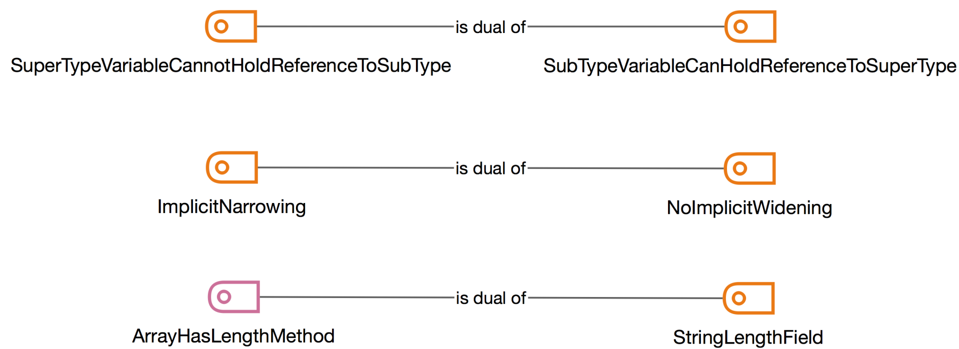


Figure 6.2. Map of dual misconceptions.

The first pair of misconceptions are about inheritance and its consequences on the type system.

SuperTypeVariableCannotHoldReferenceToSubType	
Title	SuperTypeVariableCannotHoldReferenceToSubType
Context	Class `Child` extends class `Parent` and there is a variable `parent` of type `Parent`
Description	Variable `parent` cannot contain a reference to an object of a sub type such as `parent = new Child();`
JLS	§4.10 Subtyping and §4.12.2 Variables of Reference Type
Observations	Sessions 8 and 10

SubTypeVariableCanHoldReferenceToSuperType

Title	SubTypeVariableCanHoldReferenceToSuperType
Context	Class `Child` extends class `Parent` and there is a variable `child` of type `Child`
Description	Variable `child` can contain a reference to an object of a super type such as `child = new Parent();`
JLS	§4.10 Subtyping and §4.12.2 Variables of Reference Type
Observations	Session 8

Example: in the following situation

```
public class Employee {
  ...
}
public class ProjectManager extends Employee {
  ...
}
public class Demo {
  public static void run() {
      ProjectManager alice = new Employee(100);
  }
}
```

student P5, when asked about the correctness of the assignment, claimed that one cannot store into a variable of a subtype a reference to an object of a superclass and that only the reverse is true. We speculate that maybe the student remembers that only one of the two settings is valid but cannot remember exactly which one. This could stem from not having understood *why* this design choice has been made and tried just to learn by heart.

The second pair of dual misconceptions is related to automatic “conversions” performed by the Java type system to cast between compatible data types. While all the intricacies related to the type system are on purpose completely avoided during the course, we still want students to gain basic confidence to deal with primitive types and their common usages.

IMPLICITNARROWING

Title	ImplicitNarrowing
Context	Assignment to a variable or variable initialization
Description	Narrowing is always implicitly performed to cast a wider type into a smaller one
JLS	§5.1.3 Narrowing Primitive Conversion
Observations	Session 7

Example: student P6 explained the following statement

```
float radius = 2.5;
```

as a double literal being automatic converted to fit into a float variable. Although some narrowing operations are possible in variable initialisation, one cannot (implicitly) initialise a `float` variable with a `double` literal: it is prevented to avoid a potential lossy conversion.

NoImplicitWidening

Title	NoImplicitWidening
Context	Assignment to a variable or variable initialization
Description	Widening is not implicitly performed to cast a smaller type into a wider one
JLS	§5.1.2 Widening Primitive Conversion
Observations	Sessions 7 and 8

Example: student P1 argued that in the following case

```
public class Circle {
  double radius;
  public Circle(final double radius) {
      this.radius = radius;
  }
}
public class Demo {
  public static void run() {
    float radius = 2.5f;
    new Circle(radius);
  }
}
```

the call of the constructor is invalid due to the wrong type of the parameter (`double` versus `float`), an error which is not automatically solved.

The third pair of dual misconceptions deals with "computing the length" of different things in Java. Specifically, we know that even if arrays are stored on the heap like normal objects, we cannot invoke methods on them (except for those defined in `java.lang.Object`) and we have to resort to the special field `length`. On the contrary, to get the length of a string we need to call the *method* `length()`.

ArrayHasLengthMethod

Title	ArrayHasLengthMethod
Context	An array `T array`, with `T` being an arbitrary type
Description	Array's length can be obtained calling `array.length()`
JLS	§10.2 Array Variables and §10.7 Array Members
Observations	Session 9

STRINGLENGTHFIELD

Title StringLengthField
Context `String s`
Description String's length can be obtained accessing the public field `s.length`
JLS §4.3.3 The Class String and docs of `java.lang.String`
Observations Session 9

Example: student P3 coded the `print` method, which is supposed to print the all names with at least six characters, this way:

```
public class ContactsAL {
  private ArrayList<String> names;
  ...
  public void print() {
    for (String name : names) {
      if (name.length >= 6) {
        System.out.println(name);
      }
    }
  }
}
```

6.3 Misconceptions can be caused by wrong analogies

Humans make sense of the world using analogies. Some researchers even believe that analogy is the core part of cognition [Hofstadter, 2001] and therefore plays a fundamental role in the learning process. With respect to this, we tried to select the misconceptions that can be viewed as relatively direct consequences of wrong analogies that students made. The conceptual map in Figure 6.3 shows them.

We have already discussed how to retrieve the length of an array or a `String`. We now look at other examples.

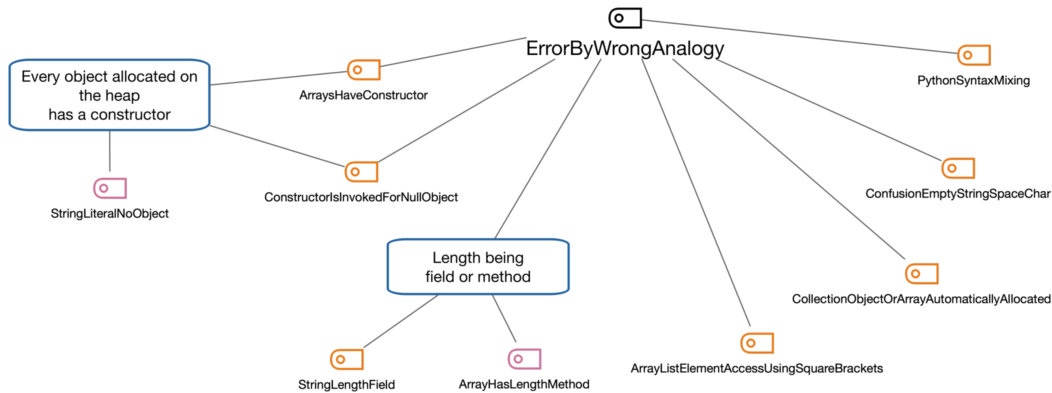


Figure 6.3. Map of misconceptions caused by wrong analogies.

ArrayListElementUsingSquareBrackets	
Title	ArrayListElementUsingSquareBrackets
Context	An `ArrayList<T> list` is defined an allocated
Description	Elements in the list can be accessed using the `[]` operator, as with arrays.
JLS	Docs of `java.util.ArrayList`
Observations	Session 9

Example: student P4 coded the `print` method this way:

```java
public class ContactsAL {
  private ArrayList<String> names = new ArrayList<>();
  ...
  public int count() {
    return names.size();
  }
  public void print() {
    for (int i = 0; i < count(); i++) {
      System.out.println(names[i]);
    }
  }
}
```

instead of using the `get()` method, explaining that he accessed that way “like we do with arrays”.

ConfusionEmptyStringSpaceChar

Title	ConfusionEmptyStringSpaceChar
Context	A situation where one deals with characters and strings.
Description	An empty `String` and the space character are the same thing.
JLS	§4.3.3 The Class String, §3.10.4 Character Literals and §3.10.5 String Literals
Observations	Session 4

Example: student P1 explained the following method:

```
public char firstChar(String s) {
  if (s != null && s.length() > 0) {
    return s.charAt(0);
  } else {
    return ' ';
  }
}
```

saying that in the `else` branch we would return an empty `String`. This confusion may originate from a not clear understanding of the difference between strings and characters or from the confusion of this method that has to deal with both. Our previous catalogue of misconceptions also contains related misconceptions, such as using single quotes for string literals, confusing strings of length one with characters or, at a higher level of abstraction, a confusion between an element and the whole collection.

The next box presents a quite pervasive problem as students at USI take the Algorithms course in parallel with the PF2 course, but the former uses Python as a programming language to code problems' solutions. Since students are not yet experienced in either of the two languages, they often end up mixing their syntax.

PythonSyntaxMixing

Title	PythonSyntaxMixing
Context	Any Java program.
Description	Fragments of the Python syntax creep into the Java source code.
JLS	Whole JLS
Observations	All sessions (evidence found in sessions 3, 4, 5, 6 and 7)

Example 1: student P4 wrote this assignment:

```
public class Car {
  ...
  private boolean breaks;
  public Car() {
    ...
    breaks = False;
  }
}
```

saying aloud that he did not remember whether false had the first letter uppercase letter (as in Python) or lower-case (as in Java).
Example 2: two students (P4 and P5) tried to print multiple variables in the following way

```
System.out.println(i, "special")
```

separating multiple parameters with a comma as one does with Python's `print` function.

Moreover, we identified that a set of misconceptions seem to have a recurring common underlying misconception about allocation: the belief that everything allocated on the heap has a corresponding constructor, which is called at its creation. The next misconception presented, which serves as an example, could theoretically also be caused by a wrong analogy with Javascript that actually has constructors for arrays[1], but as students have not been exposed to that language at this point in their curriculum, we believe that it is not necessarily due to that.

[1] Documentation for `Array()` constructors in Javascript: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Array`.

ArraysHaveConstructor

Title	ArraysHaveConstructor
Context	An array `T array`
Description	When the statement `array = new T[1];` is executed, a constructor is invoked.
JLS	§10.3 Array Creation
Observations	Session 9

Example: for the following snippet of code:

```
private String[] names;
...
names = new String[1];
```

student P1 drew this Stack and Heap diagram (note on the left column the stack frame corresponding to the array constructor):

6.4 Notional Machines can also harm

We discussed the four notional machines used in the course in Section 2.4. While the universal agreement is that notional machines *help* students learning better, this study uncovers subtle details that can reduce the efficacy of notional machines.

The big conceptual map shown in Figure 6.4 tries to group all the misconceptions that are somehow related to notional machines.

6.4.1 Students improperly use Notional Machines

Instructors, supported by research evidence, believe that notional machines are useful to their students and often “force” them to use these pedagogical devices against the will of “lazy students”. However, when students perceive the notional machine solely

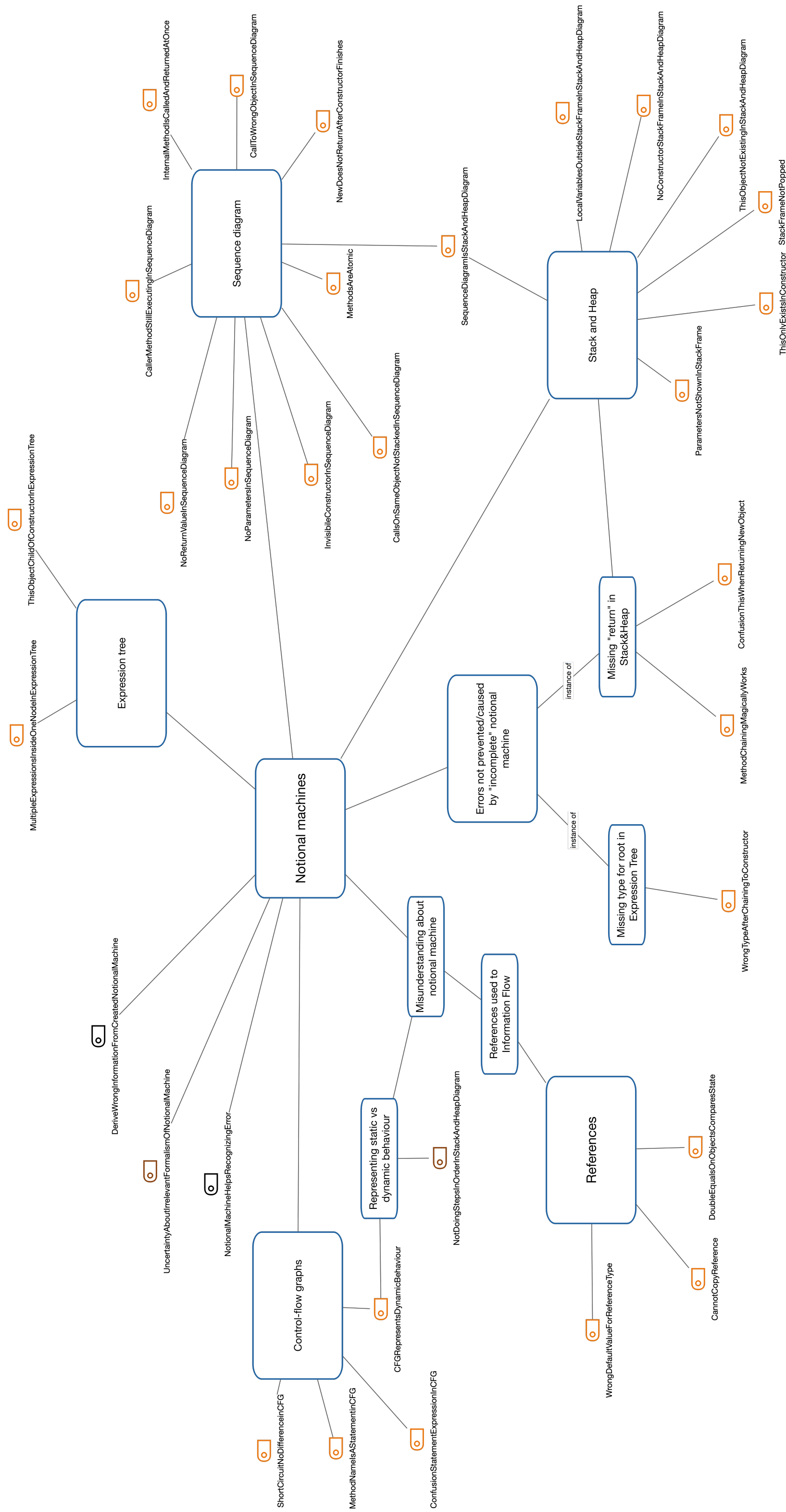


Figure 6.4. Map of codes related to notional machines.

as a boring superfluous addition to the main task, which remains programming, they may not benefit from it.

We specifically analysed how students draw the Stack and Heap diagram (see Section 2.4.1), which contains a vast set of information and therefore it usually requires a lot of time to be correctly drawn even for snippets of code of modest size. We observed several cases where they tried to use shortcuts which limited the efficacy of the notional machine.

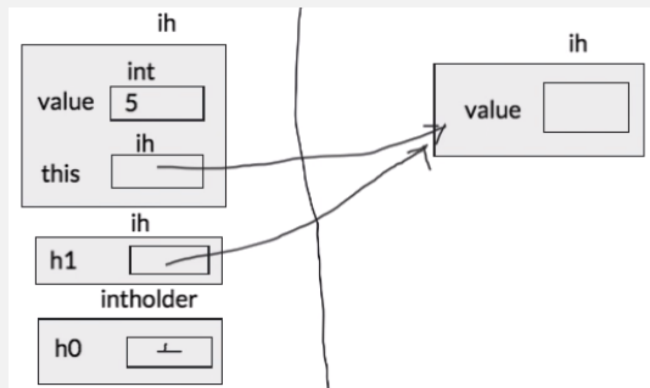
NotDoingStepsInOrderInStackAndHeapDiagram

Title	NotDoingStepsInOrderInStackAndHeapDiagram
Context	Drawing a Stack and Heap diagram to show the execution of a method.
Description	Execution steps (allocation of new stack frames, objects creation, fields initialisations, and so on) are not carried out in the exact sequence in which they happen.
JLS	N/A
Observations	Sessions 1, 2, 8, 9 and 10

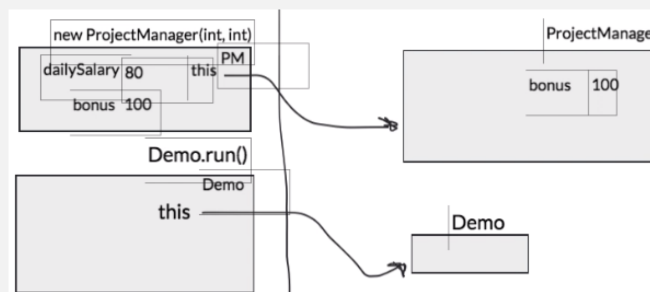
Example 1: student P2 was drawing the diagram for the following code:

```
IntHolder h0 = null;
IntHolder h1 = new IntHolder(5);
```

and before executing the body of the constructor that sets the field in the object to the value passed as a parameter, he assigned to the local variable `h1` the reference to the newly born object.



Example 2: student P4 was drawing the diagram in session 8 (see 4.8 for the code) and initialised the field `bonus` before executing the `super` call.



These examples show that students do not perceive a real one-to-one mapping between every action *they do* on the diagram and every concrete execution step performed by the *JVM*. We believe that by strictly enforcing this mapping we would not only prevent incorrect drawings but also stop early on the development of misconceptions. The rigidity of the mechanism actually impedes some "creative" ways students often employ to escape from a problem they previously made.

6.4.2 Notional Machines are incomplete

Some specific aspects of notional machines are not formalised on purpose, because their very essence lies in hiding and abstracting some of the intricacies of the *real* machine to make students more comfortable and hard topics more approachable.

However, we have found evidence that some incompleteness may "disturb" the learning process. Disturb here is meant in a broad sense: some errors might be directly caused or simply not prevented by the current form of the notional machine. We want to highlight two specific instances found in the study.

Missing type for root node in Expression Tree

In our Expression Tree notation (see 2.4.3) we often omit the type of the last "operation", which should be indicated above the root node and is the type associated to the whole expression.

We argue that this should be made very clear to students, who should be encouraged to always reason about the type of every expression and write it down on the tree. Consider for example the following misconception: we believe that by forcing students always writing types we would have prevented or weakened the presence of this error.

WrongTypeAfterChainingToConstructor

Title	WrongTypeAfterChainingToConstructor
Context	Method calls are chained to a constructor invoked creating an object with `new`.
Description	Regardless of how many methods we chain and their result types, the type of the whole expression is always the class of the newly created object.
JLS	§8.4.5 Method Result and §15.3 Type of an Expression
Observations	Sessions 1, 3, 6, and 7

Example: having this class that models an engine:

```
public class Engine() {
  ...
  public Engine() {
    ...
  }
  public void setSpeed(int speed) {
    ...
  }
}
```

students were asked to create a new engine, to set its speed to 2 and to store it in a local variable. Four out of six produced this one-line solution:

```
Engine e = new Engine().setSpeed(2);
```

without paying attention that, given the way it is designed, this class is not immutable and when you call the `setSpeed` method you get nothing back (`void`).

Improper handling of return in Stack and Heap diagram

The Stack and Heap diagram introduced in the course (see 2.4.1) is rather complete with respect to method calls which frames are allocated on the stack and to objects which are allocated on the heap. However, observations in this study have shown that a subtle detail of the execution of a program is not handled properly, meaning that it is not completely clear what one should do when the situation occurs.

This problem is about the return value of non-void methods. The current notation lacks a proper place to draw *where* that value is stored after popping the stack frame and before an eventual assignment to a local variable in the caller method. This does not usually constitute a problem, as the two moments occur one right after the other.

However, in complex expressions such as ones with nested method calls, only the final result is assigned to a local variable, while the intermediate result of the nested method call is passed as a parameter to the outer method. Or, as another example, in chained method calls, the final result is assigned to the local variable only at the end of the execution of the whole chain, and intermediate results are used as objects on which to invoke subsequent methods.

In the course it is usually explained that "we hold in our hand" the result and sometimes a small box is drawn on the stack near the popped frame to contain it. This is not strictly enforced and maintained throughout the semester and may instil uncertainty.

We present two misconceptions we believe may be fixed defining and making clear how to handle the return value.

METHODCHAININGMAGICALLYWORKS

Title	MethodChainingMagicallyWorks
Context	A chain of method calls: `o.m().n()`
Description	Chaining works through "magic" assistance by the JVM and not because each method is called on the result of the previous one.
JLS	§15.12 Method Invocation Expressions
Observations	Session 9

Example: in this context:

```
public class Contacts {
  ...
  public void addName(String name) {
    ...
  }
}

public class Demo {
  public static void run() {
    new Contacts().addName("Luca");
```

two students (P3 and P4) could not explain how could it be that `addName` has the correct reference to the `Contacts` object in the `this` keyword, claiming that "maybe the `run` method keeps somehow track of things" and "I'm not sure because [the return value] is not stored as a variable".

ConfusionThisWhenReturningNewObject

Title	ConfusionThisWhenReturningNewObject
Context	With immutability, a new object is returned after invoking a method that mutates the existing object.
Description	When using an immutable design, `this` within method invocation references a different (new) object in the heap.
JLS	§15.12 Method Invocation Expressions
Observations	Session 6

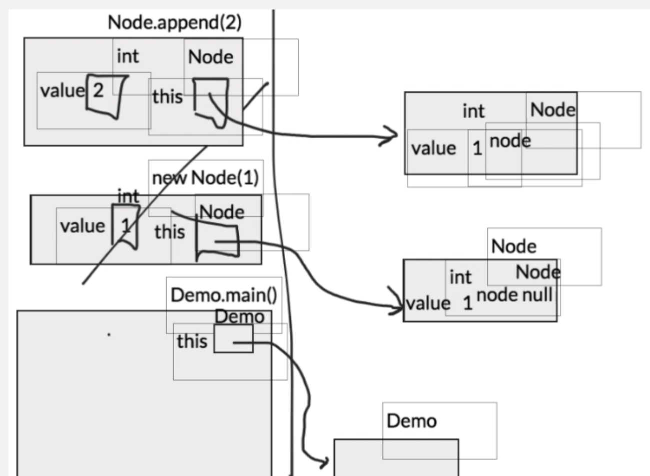
Example: with the following design of an immutable `Node` class as an element for building lists:

```
public class Node {
  private int value;
  private Node tail;
  public Node(final int value) {
    this.value = value;
    this.tail = null;
  }
  public Node(final int value, final Node tail) {
    this.value = value;
    this.tail = tail;
  }
  public Node append(final int value) {
    return new Node(this.value, new Node(value, this.tail));
  }
  ...
}
```

student P4, when asked to draw a Stack and Heap diagram showing the execution of the expression

```
new Node(1).append(2)
```

misunderstood the method chaining and instead created a new object to make the `this` of the `append` method "pointing to a new object, since the class is immutable".



The misconceptions we have just presented target two very narrow and specific aspects of our notional machines. Nonetheless, we believe that putting the accent on them is beneficial for researchers who ideate new notional machines and for teachers to be aware of the pitfalls students may subtly encounter when using these pedagogical devices.

6.5 Tackling a problem the right way is hard

Even after learning the basics of syntax and semantics of a programming language, many students struggle with coming up on their own with an *algorithmic* solution for a given problem. The task does not even need to be intrinsically difficult: the famous “Rainfall problem” presented at the beginning of Chapter 1 would be certainly classified as a *simple* problem and yet it constitutes a barrier for many.

In [Davies, 1993], Davies observes that the part of literature that deals with the “analysis of the strategies commonly employed by programmers in the generation and the comprehension of programs” is limited but is crucial to integrate the “knowledge representation” with a “strategic model”.

We collected in the map shown in Figure 6.5 the main codes taken from the StrategyErrors category (see Section 5.2); they highlight issues shown by participants that led to tackling a problem the wrong way.

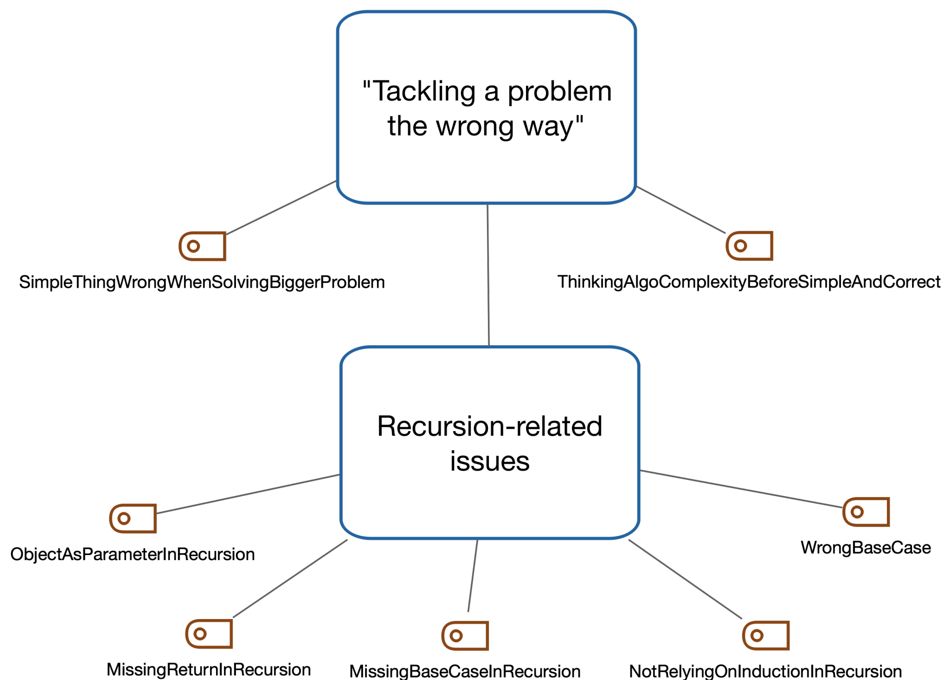


Figure 6.5. Map of codes related to how to tackle problems.

We describe here two codes that can be applied to generic problems and we devote Section 6.5.1 to present the ones that are specific to recursive solutions.

SimpleThingWrongWhenSolvingBiggerProblem

Title	SimpleThingWrongWhenSolvingBiggerProblem
Context	A generic problem to solve
Description	When solving a problem, one has to think at multiple levels of abstraction, which means dealing with the "big picture" which presents the problem as a whole and with the small details which must however be carried out correctly. When confronted with a significant challenge, students' cognitive load is so devoted to the big problem that they have no resource left to pay attention to the smaller pieces and fail to solve them properly.
JLS	Not applicable
Observations	All sessions

Example 1: while implementing a recursive method that should find the minimum over an array, student P1 used the single equal sign inside a comparison:

```
public static int findMinRec(int[] array, int pos) {
  if (array.length = 1) {
    ...
  }
}
```

In no other occasion the student did the same mistake and thus we safely claim that he has not a misconception related to the meaning of the single equal operator versus the double equal operator.

Example 2: while implementing a recursive method to sum the elements in a list, student P1 called a no-parameter method without using parentheses:

```
public class Node {
  ...
  public int sum() {
    return this.value + this.node.sum;
  }
}
```

Again, this was the only time he made this specific mistake, probably overloaded by thinking about how to solve the problem recursively.

THINKINGALGOCOMPLEXITYBEFORESIMPLEANDCORRECT	
Title	ThinkingAlgoComplexityBeforeSimpleAndCorrect
Context	A problem that admits advanced solutions with low computational complexity.
Description	Most of the problems that can be solved in simple ways also admit solutions with lower computational complexity, which unfortunately are often not trivial. Novices seek these kinds of solutions, above their competence level, *before* coding a simple and correct one.
JLS	Not applicable
Observations	All sessions
Example: student P1 could not come up with a valid recursive solution to find the minimum over an array. When he received hints on how to proceed, he realised that he was overthinking and "thinking about how to lower the complexity starting in the middle and...".	

The fact that these thoughts are present should come as no surprise: students at USI attend this course and the Algorithms one in parallel and it is thus easier to mix things. However, many instructors are not aware of subtle details like this one, because they see students taking their course in isolation, while this is not the case.

6.5.1 Recursion problems are especially hard

We noticed that students struggled significantly when confronted with problems they had to solve recursively. While it is acknowledged that "thinking recursively" is not trivial, this was a relative surprise as our students, during the first programming course in the Bachelor program, learn *functional* programming.

Literature contains interesting prior work on this. Colleen, using interviews in a qualitative study similar to ours [Lewis], has analysed four substitution techniques students often employ to trace linear recursion. Hamouda [Hamouda et al.] built a concept inventory that is specific to recursion problems. We note that some of the misconceptions we recognised are the same of some she identified. In detail:

- Our MISSINGRETURNINRECURSION maps to her RCNORETURNREQUIRED (return is not required for "backward flow");
- Our MISSINGBASECASEINRECURSION and WRONGBASECASEINRECURSION loosely map to her BCWRITE (difficulty to write a base case);
- Our NOTRELYINGONINDUCTIONFORRECURSION loosely maps to her RCWRITE (difficulty to write a proper recursive call).

Moreover, research has been carried out to understand how prior knowledge (in this case, having learnt functional programming with a specific programming language) affects learning of new concepts (different programming paradigms in Java). In [Santos et al., 2019] four specific problems detected among students at USI when transitioning from Racket to Java are discussed.

In this work, we want to present some pieces of evidence we have found while analysing and coding the sessions' recordings that show a great number of "strategical" aspects one has to master to solve recursively a problem.

OBJECTASPARAMETERINRECURSION

Title	ObjectAsParameterInRecursion
Context	A method that performs any kind of structural recursion.
Description	When a method that belongs to a class which represents an element in a recursive data structure needs to compute a value using recursion, we need to pass the object as a parameter of the method.
JLS	Not applicable
Observations	Session 6

Example: given a `Node` class that represents an element of a linked list, two students (P1 and P5) attempted to implement a `sum` method, which is supposed to return the sum of all the elements in the list when invoked on the first element, passing the "current" node as a parameter:

```
public class Node {
  ...
  public int sum(Node node) {
    ...
  }
}
```

They probably did not realise that one already has access to the object on which the method is called each time using `this`. Note also that this example comes from session 6, a point in the course at which students are already expected to have familiarity with classes and objects that are introduced in the very first week and are practised in all the labs.

MissingReturnInRecursion

Title	MissingReturnInRecursion
Context	A method that performs any kind of recursive computation.
Description	In the recursive case of a recursive method, one calls the same method with different parameters, optionally does other computations, but forgets to return the result.
JLS	Not applicable
Observations	Sessions 5 and 6

Example: given a `Node` class that represents an element of a linked list, student P1 implemented the `indexOf` method, which is supposed to return the position at which a value is found in the list or -1 if it is not present, in the following way:

```
public class Node {
  private final int value;
  private final Node node;
  public int indexOf(int v, int index) {
    if (node == null) {
      return -1;
    } else if (v == value) {
      return index;
    }
    node.indexOf(v, index + 1);
  }
}
```

Keeping aside the bug while checking the last element, the important piece here is that the resulting value of the call to `node.indexOf()` in the last statement is ignored and not returned, preventing the recursion to give back a result.

Errors such as MissingReturnInRecursion can be prevented using an IDE that ensures that the method always returns a value. The automatic mechanism is definitely helpful to catch the bug earlier and to prevent wasting a lot of time in search of it, but it is questionable whether it is really beneficial to a learner who should understand exactly why that `return` is necessary and what would happen if one leaves it out.

MissingBaseCaseInRecursion

Title	MissingBaseCaseInRecursion
Context	A method that performs any kind of recursive computation.
Description	The recursive method is implemented without a base case that stops the recursion.
JLS	Not applicable
Observations	Sessions 5 and 6

Example: given a `Node` class that represents an element of a linked list, student P1 implemented the `sum` method, which is supposed to return the sum of all the elements in the list when invoked on the first element, without a base case in the following way:

```
public class Node {
  private final int value;
  private final Node node;
  public int sum() {
    return this.value + this.node.sum();
  }
}
```

Students should be made aware that it never makes sense to not have a base case and should be pushed to always think about it when they start writing a recursive method.

WrongBaseCaseInRecursion

Title	WrongBaseCaseInRecursion
Context	A method that performs any kind of recursive computation.
Description	The condition that defines the base case of a recursive method is wrong.
JLS	Not applicable
Observations	Sessions 5 and 6

Example 1: to recursively compute the minimum over an array, student P3 defined this condition:

```
public static int findMinRec(int[] array, int pos) {
  if (pos == 0) {
    ...
  }
  ...
}
```

which cannot work as the original call was `findMinRec(array, 0)`.
Example 2: in the same problem, a student wanted to make the condition for the base case “when you have found the minimum”.

NotRelyingOnInductionForRecursion

Title	NotRelyingOnInductionForRecursion
Context	A method that performs any kind of recursive computation.
Description	While implementing a recursive method one does not rely on the fact that the next recursive call gives back the desired value, often without yet knowing what that value could be.
JLS	Not applicable
Observations	Sessions 5 and 6

Example: when trying to recursively find the minimum over an array, one of the possible implementations is to define a method which returns the minimum from the "position it is called on" to the end of the array. The base case would be being at the last position, at which point the minimum is trivially the last element itself. Two students (P1 and P5) faced insurmountable problems when trying to implement the solution, even after identifying and coding the base case correctly:

```
public static int findMinRec(int[] array, int pos) {
  if (pos == array.length) {
    return array[pos];
  } else {
    // ?
  }
}
```

They admitted to being confused: "I don't know where to store the minimum" or "So now I should compare this position with position 0". They seemed lost in general, but we believe that one key missing piece in their reasoning was that they did not rely on the fact that the "next" recursive call provides the correct answer to a subproblem. Thinking recursively, as mathematical induction, has an intrinsically high cognitive load and therefore it is essential to assume that the inductive step works to focus on the proper definition of the "current" step.

Chapter 7

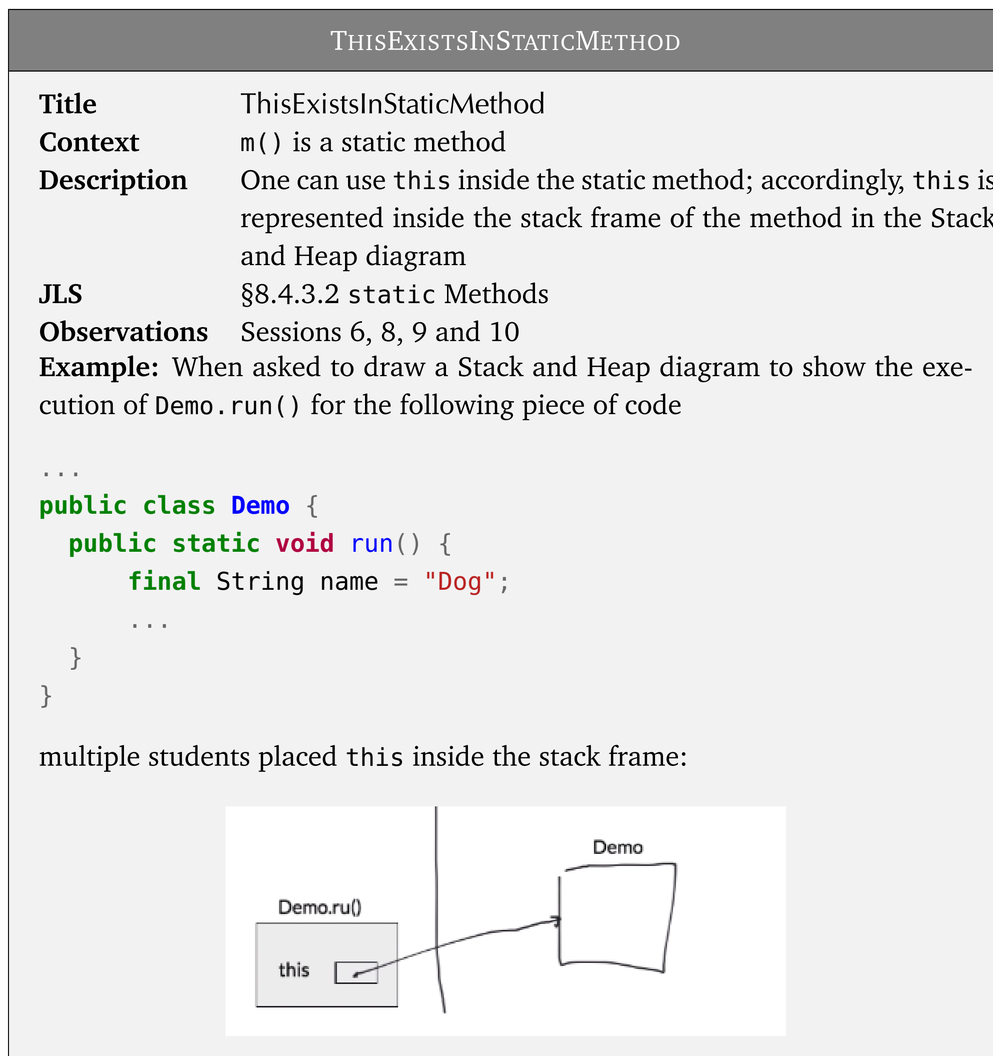
Insights about learning trajectories

diSessa analysed the history of research on conceptual change and found that "shockingly, almost no research tracks students' moment-by-moment thinking while learning" [diSessa et al., 2014]. This study is an opportunity, as it has followed the same six students for ten weeks distributed across the whole course while they were continuously learning new concepts.

In this work we move the first steps in the direction of looking at the evolution of learning, looking not only at the specific points in which students demonstrate misconceptions, but also at their *learning trajectory* to capture at which pace, if ever, their understanding improves, their mental models get fixed or, perhaps, if there is a recrudescence and some topics that were considered acquired forever show symptoms of problems later in time.

7.1 Misconceptions persist over time if not corrected

We have found evidence that some misconceptions persist until the end of the course if nobody (instructor or peers) corrects the flaw. Consider the following misconception.



THISEXISTSINSTATICMETHOD

Title	ThisExistsInStaticMethod
Context	`m()` is a static method
Description	One can use `this` inside the static method; accordingly, `this` is represented inside the stack frame of the method in the Stack and Heap diagram
JLS	§8.4.3.2 `static` Methods
Observations	Sessions 6, 8, 9 and 10

Example: When asked to draw a Stack and Heap diagram to show the execution of `Demo.run()` for the following piece of code

```
...
public class Demo {
  public static void run() {
      final String name = "Dog";
      ...
  }
}
```

multiple students placed `this` inside the stack frame:

We show in Table 7.1 the assessment of correctness for five participants[1] in the four sessions that targeted this misconception. Participants' identifiers are assigned at random.

The panoramic offered by Table 7.1 is remarkable in its simplicity, as it shows how even a very tiny and easily-correctable misconception persists in time if not corrected. While one could judge this as a minor detail, we claim that misconceptions are often intertwined and big errors often have their foundations in small ones.

Consider now a completely different but common misconception among novices who learn about class inheritance for the first time.

[1]The sixth participant was not available during those sessions and has been excluded from the table.

Session	P1	P2	P3	P4	P5
Session 6	Correct	Wrong	N/A	Wrong	N/A
Session 8	Wrong	Wrong	Wrong	Wrong	Wrong
Session 9	Wrong	Wrong	Wrong	Correct	Wrong
Session 10	Wrong	Wrong	Wrong	Wrong	Wrong

Table 7.1. Correctness of ThisExistsInStaticMethod across four sessions.

SuperclassObjectIsAllocated

Title SuperclassObjectIsAllocated

Context Class `Child` extends class `Parent`

Description When `new Child()` is executed, two objects are created: a `Child` objects with the fields that belong to the class `Child` and a `Parent` object with the fields that belong to the class `Parent`.

JLS §8.2 Class Members

Observations Sessions 8 and 10

Example: Assuming the following class hierarchy:

```java
public class Empolyee {
  private int dailySalary;
  ...
}

public class ProjectManager extends Empolyee {
  private int bonus;
  ...
}
```

multiple students, when asked to draw a Stack and Heap diagram to show the execution of `new Employee()`, created two separate objects on the heap:

We track this misconception over two sessions in Table 7.2.

Session	P1	P2	P3	P4	P5
Session 8	Correct	Correct	Wrong	Wrong	Wrong
Session 10	Correct	Correct*	Correct	Correct	Correct

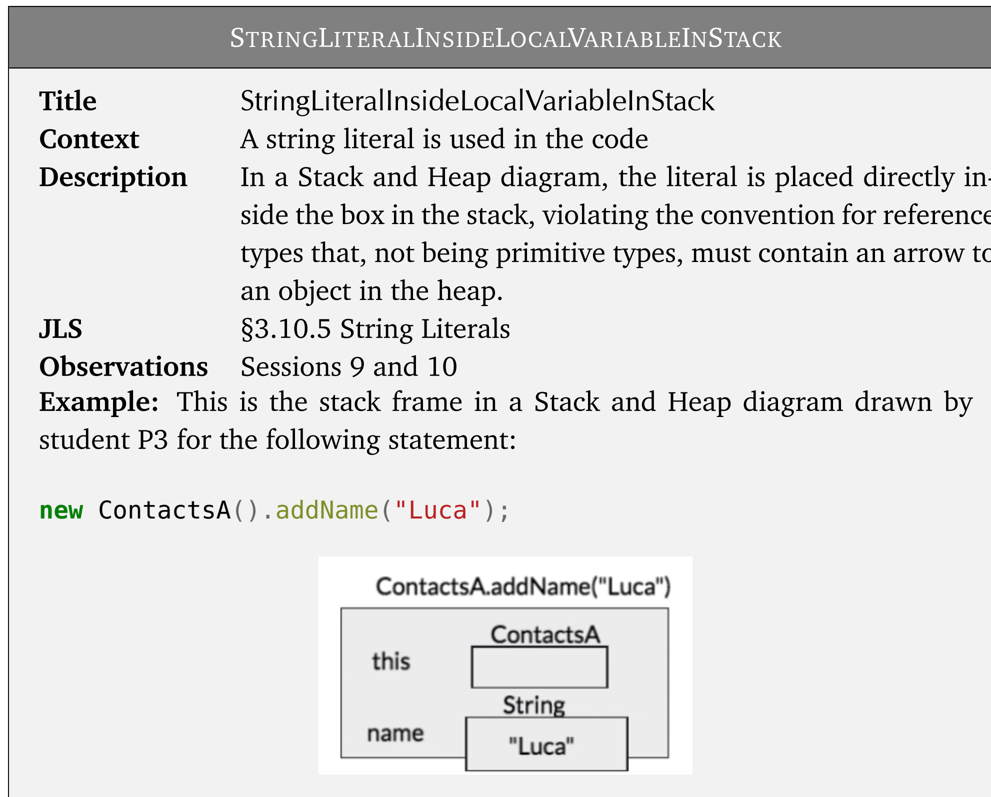
Table 7.2. Correctness of SuperclassObjectIsAllocated across two sessions.

The asterisk marks a minor detail for participant P2, who showed uncertainty on how to handle inheritance with sequence diagram. His Stack and Heap diagram was correct, demonstrating mastery on the topic, and thus the issue with the sequence diagram could be a symptom of a notional machine not well formalised, a problem already tackled in Section 6.4.2.

What happened between the two sessions? We (I and professor Hauswirth) routinely discuss the insights that gradually become available as the study progresses. The importance of doing this is twofold: from one side, we can plan future mastery check sessions better, choosing topics wisely; on the other, as the professor is also in charge of teaching the course, he can steer it and explain again topics that evidence shows are not well understood. Thus, this specific misconception was explicitly addressed and the correct concept has been explained during a class session in the course held in the middle of the two observation sessions. Even if that has been a single intervention, it sufficed to fix the flaw for all the participants.

This proves that timely feedback is crucial for an effective teaching intervention and remarks the necessity to frequently collect information about students' progress in mastering the concepts without relying on assumptions made by the instructor about what they have actually learnt.

If we look at other examples, we can see that our observations to detect misconceptions are reasonably stable. Consider the following case.



STRINGLITERALINSIDELOCALVARIABLEINSTACK

Title StringLiteralInsideLocalVariableInStack
Context A string literal is used in the code
Description In a Stack and Heap diagram, the literal is placed directly inside the box in the stack, violating the convention for reference types that, not being primitive types, must contain an arrow to an object in the heap.
JLS §3.10.5 String Literals
Observations Sessions 9 and 10
Example: This is the stack frame in a Stack and Heap diagram drawn by student P3 for the following statement:

```
new ContactsA().addName("Luca");
```


Table 7.3 tracks the assessment for five participants and shows that most of them already did not have the misconception in session 9 (this is good, since it was near the end of the course). Without specific teaching interventions, one student did not correct his error, while another managed to provide a correct solution in session 10. This case shows that is not mandatory to have teaching interventions in class to fix issues, but rather than these fixes can arise from a great variety of sources: peers during group study sessions, private discussions with teaching assistant, or even indirect help from studying a seemingly unrelated topic that helped to remove confusion.

Session	P1	P2	P3	P4	P5
Session 9	Correct	Correct	Wrong	Correct	Wrong
Session 10	Correct	Correct	Wrong	Correct	Correct

Table 7.3. Correctness of StringLiteralInsideLocalVariableInStack across two sessions.

The feedback thematic is also explored in chapter five of [Ambrose et al., 2010], which brings further research evidence of its importance. “The power of feedback” [Hattie and Timperley, 2007] is another famous paper from the educational research area that has an eloquent title and systematically reviews the influence of feedback on learning.

Chapter 8

Conclusions and follow up studies

This qualitative study constitutes the first milestone of the *exploratory stage* for the bigger research project in our group. Even if not systematically, we have tried to map behaviours and artifacts, namely drawings of notional machines and pieces of source code, to specific Java concepts and to strategies used to solve problems.

One important outcome of this work is the recordings of the ten mastery check sessions held with students, which have been stored securely on a NAS. Their future use in other studies is permitted but precisely limited, according to the privacy policy described in Section 3.3.

Videos are now tagged with about 4 000 codes that enable a number of more targeted future studies. We currently envision the following possibilities:

- We significantly expanded the pre-existing set of Java misconceptions with more than a hundred new ones. They can now be added to the curated inventory of programming misconceptions maintained by our research group[1].
- Problems related to notional machines may now be studied in isolation and thoroughly. Although in this study we have covered all the four notional machines used in the course, research literature currently lacks in-depth studies about the potential of notional machines and comparisons among them. We have uncovered issues that may arise when a teacher uses them without care; therefore it is essential to review and emphasise these problems before broadening their usage to secondary and possibly primary education.
- Studying learning trajectories is widely considered difficult, but we have presented a first simple way to assess them. In fact, the key resides in tracking students' mastery levels on the same topic at multiple moments in time, and augmenting it with information about the teaching interventions made in the middle of those snapshots. Without these details one risks to end up speculating about

[1] `https://progmiscon.org/` is a prototype website that contains a preview of misconceptions collected in the past by the group.

what worked and what did not, instead of formulating reasonably valid hypotheses.

- While we added codes for tracking students' psychological attitude and meta information about the different phases of the interview, we did not fully exploit this kind of data in our analyses. One could work in close contact with people more knowledgeable of cognitive sciences and psychology to characterise specific moments of a mastery check session and see how they influence the "performance". For instance, nervous students might fail to correctly answer just because they do not feel at ease during the interview. Or, as another example, it might be interesting to study to what extent and for which students asking a question twice, that implies putting the accent on something, makes them recognise a mistake.

In addition, this work contributes to the more general field of qualitative and mixed-method research with the development of a new powerful tool (Section 3.5) to create an edited video from multiple sources with automatic captioning of the voices. While in the current state this piece of software needs expert knowledge to be used, more user-friendly interfaces (such as a GUI) could be developed on top of it.

Bibliography

Susan A Ambrose, Michael W Bridges, Michele DiPietro, Marsha C Lovett, and Marie K Norman. *How learning works: Seven research-based principles for smart teaching*. John Wiley & Sons, 2010.

David John Barnes, Michael Kölling, and James Gosling. *Objects First with Java: A practical introduction using BlueJ*. Pearson/Prentice Hall, 2006.

John Biggs. Enhancing teaching through constructive alignment. *Higher education*, 32 (3):347–364, 1996.

Benjamin S Bloom. Learning for mastery. instruction and curriculum. regional education laboratory for the carolinas and virginia, topical papers and reprints, number 1. *Evaluation comment*, 1(2):n2, 1968.

Denis Bogdanas and Grigore Roşu. K-java: a complete semantics of java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 445–456, 2015.

Clark A Chinn and Bruce L Sherin. Microgenetic methods. 2014.

Simon P Davies. Models and theories of programming strategy. *International Journal of Man-Machine Studies*, 39(2):237–267, 1993.

Andrea A diSessa et al. A history of conceptual change research: threads and fault lines. In *The cambridge handbook of the learning sciences*, pages 88–108, 2014.

Benedict Du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73, 1986.

Thomas R Guskey. Lessons of mastery learning. *Educational leadership*, 68(2):52, 2010.

Mark Guzdial. Learner-centered design of computing education: Research on computing for everyone. *Synthesis Lectures on Human-Centered Informatics*, 8(6):1–165, 2015.

Sally Hamouda, Stephen H. Edwards, Hicham G. Elmongui, Jeremy V. Ernst, and Clifford A. Shaffer. A basic recursion concept inventory. 27(2):121–148. ISSN 0899-3408, 1744-5175. doi: 10.1080/08993408.2017.1414728. URL `https://www.tandfonline.com/doi/full/10.1080/08993408.2017.1414728`.

John Hattie and Helen Timperley. The power of feedback. *Review of educational research*, 77(1):81–112, 2007.

Matthias Hauswirth and Andrea Adamoli. Identifying misconceptions with active recall in a blended learning system. In *European Conference on Technology Enhanced Learning*, pages 416–421. Springer, 2017.

Douglas R Hofstadter. Analogy as the core of cognition. *The analogical mind: Perspectives from cognitive science*, pages 499–538, 2001.

Jeffrey D Karpicke and Henry L Roediger. The critical importance of retrieval for learning. *science*, 319(5865):966–968, 2008.

Colleen M. Lewis. Exploring variation in students' correct traces of linear recursion. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 67–74. Association for Computing Machinery. ISBN 978-1-4503-2755-8. doi: 10.1145/2632320.2632355. URL `https://doi.org/10.1145/2632320.2632355`.

Colleen M Lewis. The importance of students' attention to program state: a case study of debugging behavior. In *Proceedings of the ninth annual international conference on International computing education research*, pages 127–134, 2012.

Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 125–180. 2001.

Lance A Miller. Programming by non-programmers. *International Journal of Man-Machine Studies*, 1974.

Katarina Pantic, Deborah A Fields, and Lisa Quirke. Studying situated learning in a constructionist programming camp: A multimethod microgenetic analysis of one girl's learning pathway. In *Proceedings of the The 15th International Conference on Interaction Design and Children*, pages 428–439, 2016.

Igor Moreno Santos, Matthias Hauswirth, and Nathaniel Nystrom. Experiences in bridging from functional to object-oriented programming. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, pages 36–40, 2019.

Michael Schneider and Elsbeth Stern. The developmental relations between conceptual and procedural knowledge: A multimethod approach. *Developmental psychology*, 46 (1):178, 2010.

Judy Sheard, S Simon, Margaret Hamilton, and Jan Lönnberg. Analysis of research into the teaching and learning of programming. In *Proceedings of the fifth international workshop on Computing education research workshop*, pages 93–104, 2009.

Elliot Soloway. Learning to program= learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, 1986.

Juha Sorva et al. *Visual program simulation in introductory programming education*. Aalto University, 2012.

Elsbeth Stern. Knowledge restructuring as a powerful mechanism of cognitive development: How to lay an early foundation for conceptual understanding in formal domains. In *BJEP Monograph Series II, Number 3-Pedagogy-Teaching for Learning*, volume 155, pages 155–170. British Psychological Society, 2005.

Ian Utting, Allison Elliott Tew, Mike McCracken, Lynda Thomas, Dennis Bouvier, Roger Frye, James Paterson, Michael Caspersen, Yifat Ben-David Kolikant, Juha Sorva, et al. A fresh look at novice programmers' performance and their teachers' expectations. In *Proceedings of the ITiCSE working group reports conference on Innovation and technology in computer science education-working group reports*, pages 15–32, 2013.

Tobias Wrigstad and Elias Castegren. Mastery learning-like teaching with achievements. In *SPLASH 2017, October 22-27, 2017, Vancouver.*, 2017.