JOHANNES KEPLER
UNIVERSITY LINZ

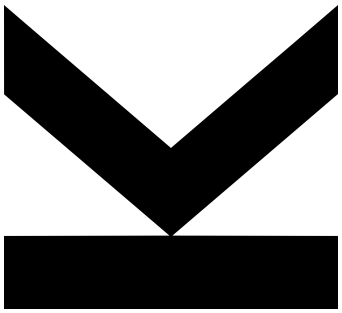
Author
Sarah Gastner
k12008166

Submission
Institute for System Software

Thesis Supervisor
o.Univ.-Prof. Dr. Dr.h.c.
Hanspeter Mössenböck

August 2023

Extension of MicroJava with Object Orientation and Exception Handling



Bachelor Thesis

to obtain the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Kurzfassung

MicroJava ist eine kleine Beispielsprache, die als Kompilierungsquelle im Compilerbau Einführungskurs an der JKU verwendet wird. Sie bietet nur sehr einfache Datentypen und einige wenige Arten von Statements an. Dadurch bleibt MicroJava eine kleine und kompakte Sprache, allerdings beschränkt das auch die Möglichkeiten der Sprache. Ziel dieser Arbeit ist es, MicroJava um Objektorientierung, inklusive Klassenmethoden, Vererbung, dynamischer Bindung, Typchecks und Type Casts, sowie Ausnahmebehandlung zu erweitern.

Abstract

MicroJava is a small sample language that is used as a compilation source in the introductory compiler construction course at the JKU. It offers only very basic data types and some basic kinds of statements. This keeps the size of the language small, but also severely restricts the capabilities of the language. The goal of this thesis is to extend MicroJava with object-orientation, including class methods, inheritance, dynamic binding, type checks, and type casts, as well as exception handling.

Contents

List of Figures

List of Tables

1. Introduction

MicroJava is a programming language used as a teaching language in lecture and exercise classes at the Institute for System Software. As the name suggests, MicroJava is inspired by the programming language Java in its syntax and semantics. However, as a teaching language, MicroJava contains only a small subset of the features Java has. This simplifies the language, but also greatly restricts the capabilities of MicroJava.

```
program Fib
  int prev1, prev2, cur;
{
  void main() {
    prev1 = 0; prev2 = 1; cur = 0;
    while (cur < 20) {
      cur = prev1 + prev2;
      print(cur); print(' ');
      prev1 = prev2; prev2 = cur;
    }
  }
}
```

Listing 1: Example MicroJava program printing the first few Fibonacci numbers.

The goal of this project is to extend the MicroJava language with two important features of Java: object orientation and exception handling. In Chapter 2, we go over the capabilities of MicroJava and the details of the proposed features. Chapter 3 explains how the features were implemented in the MicroJava compiler and interpreter and what difficulties we faced during the implementation. Chapter 4 shows how the correctness of the implementation was verified through the implementation of new unit test suites. In Chapter 5, we summarize the changes and discuss features that could be implemented in the future.

2. Background

This chapter gives an overview of the state of the MicroJava language as of before this project, as well as the new features that we implemented, object orientation and exception handling.

2.1. MicroJava

The core of this project is the MicroJava language. It is used by the lecturers of the Institute for System Software as a teaching language. Specifically, it is used in the compiler construction course, where the concepts of compilers are taught using MicroJava as a sample language. Since the course is an introductory course, MicroJava contains only those features that are relevant to the course.

This is a list of the major features and restrictions of MicroJava in the state before the extensions made in this project:

- A MicroJava program is contained within a single file.
- A program may declare global (static) variables, constants, methods and classes. They must be declared before they are used. A variable is either of a data type or a reference type.
- There are two data types: integers (`int`) and ASCII characters (`char`). Constants can only be of one of these types.
- Reference types can be arrays and classes. Arrays must be one dimensional, classes may have fields, but no methods.

- Methods may have parameters, local variables and a return type. Local variables must be declared before the beginning of the method block. The return type must be a data type or `void`.
- There must be a `main` method, which is the entry point of the program and which must have no parameters and return `void`.
- There is no garbage collector. Allocated objects live until the termination of the program, even if they are not accessible.

To complete this project, the lecturers of the compiler construction course provided their sample solution compiler and interpreter to us. They are written in Java, which is why the extensions to them are also written in Java. The architecture of the MicroJava virtual machine (VM) is similar to the Java VM, but also simplified. We will go into detail about the MicroJava VM in Chapter 3.

The compiler uses recursive descent parsing, a technique that builds a syntax tree from top to bottom using one look-ahead token. This requires the grammar of MicroJava to be *LL(1)*, which means that the production that is currently parsed can be identified from *L*eft to right with *L*eftmost derivations using *1* look-ahead token. In other words, the parser has to be able to choose an alternative in the grammar based on one look-ahead token. [1]

2.2. Object orientation

Object-oriented programming is a paradigm whose main focus lies on objects as instances of a class. A class defines the structure of an object, like data fields or methods. Through inheritance, it is possible to abstract similar structures from multiple classes into a single superclass, which reduces code duplication and increases reusability. Many features of object orientation are built upon inheritance. Those that are relevant to this project are mentioned in this section. [2]

Typical examples used for showing object orientation are shapes. Suppose there are different kinds of shapes like circles, triangles and rectangles in a graphical editor. All shapes have some common traits, like area or circumference, but some shapes have unique traits that other shapes do not have. A circle, for example, is defined by a center point and

a radius, while a rectangle is defined by four points.[1] Instead of defining the common structures in every class of a shape, these can be abstracted into a superclass, from which the concrete shapes inherit these structures, as shown in Figure 2.1. It is also possible to define methods but leave the implementation to the subclasses. This makes sense for the `draw()` method, since one cannot draw a shape without knowing the necessary details of a shape that only the subclasses provide.

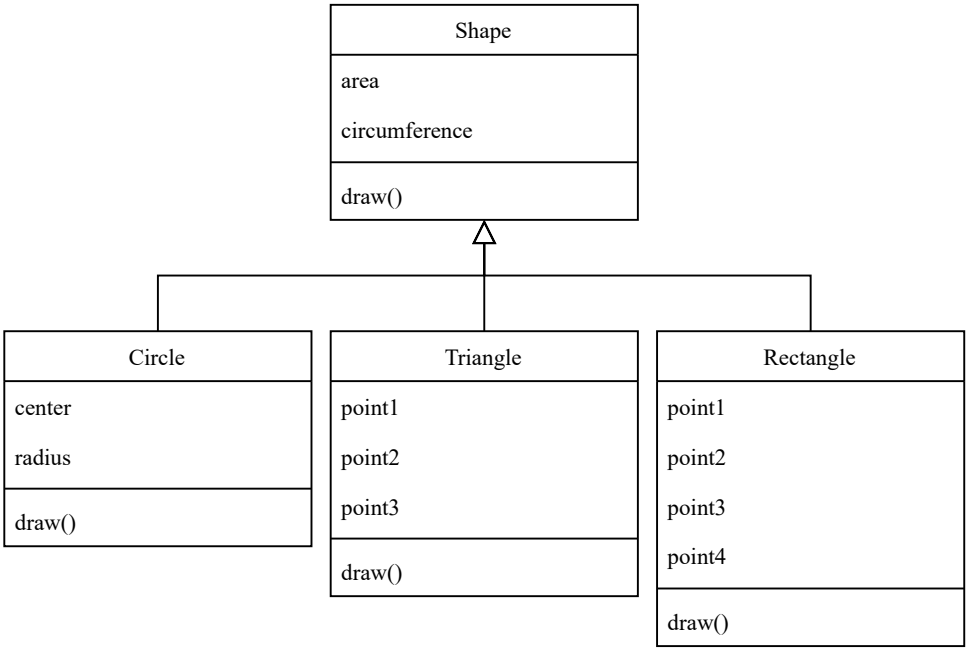


Figure 2.1.: UML class diagram of the shape example

Before the implementation of the extensions, MicroJava did allow for the definition of classes. However, inheritance was not supported in MicroJava, nor were methods in classes. One goal of this project is to implement inheritance so that classes inherit fields and methods from their superclass. Additionally, we implemented overwriting methods in subclasses and virtual method calls, i.e., determining which method is called depending on the dynamic type of an object during run-time, as well as type checking and type casting.

[1] These are not the strict mathematical definitions or the only definitions of those shapes, but just example definitions that could be used in a very simple program.

2.3. Exception handling

Any sufficiently complex program has to account for problems that might occur during run-time. A concept commonly used to catch these problems is *exception handling*. An exception is raised during the execution of the program if performing some operation leads to a condition that requires the attention of the invoker of the operation. Such a condition could be a division by zero or an error that happened while reading a file. The invoker can then handle the exception and respond to the condition using the information provided by the exception. [3]

MicroJava did not feature an exception handling mechanism. As part of this project, we implemented an exception handling mechanism that allows us to `throw` exceptions and `catch` them in `try-catch` blocks. The mechanism also makes use of the object orientation features described in Section 2.2: Thrown exceptions are caught by catch clauses whose catch parameter is of the same class or a superclass of the exception class, as is the case in Java.

3. Implementation

This chapter explains how the new features were implemented. We will show all changes to the MicroJava grammar, the compiler, and the interpreter, mostly in the order the changes were implemented. The chapter is split into two sections, Section 3.1 and Section 3.2, which contain the two main features, object orientation and exception handling, respectively.

While not the main focus of the compiler construction course, the lecture provided a sketch of the changes needed to implement the object orientation and exception handling features. Many parts of the sketch could be implemented without major changes, but we also faced unexpected difficulties during the implementation of some sections.

3.1. Object orientation

The object orientation features were the largest part of this project. The features required substantial changes to the MicroJava grammar, the compiler, and the interpreter.

Starting with the grammar, the keywords `extends` and `instanceof` were added to the existing keywords. Additionally, the following productions of the MicroJava grammar were changed as part of the implementation of the object orientation features (changes marked in red):

- `ClassDecl = "class" ident [ "extends" Type ]`
  `"{" { VarDecl | MethodDecl } "}".`
  The option `"extends" Type` allows a class to be derived from a superclass. The alternative `| MethodDecl` allows for method declarations in classes. The alternative creates an LL(1) conflict, since both alternatives can start with an identifier.

- `CondFact = Expr ( Relop Expr | "instanceof" Type ).`
  This new alternative enables type checks in conditions.
- ```
  Factor = Designator [ ActPars ]
  | number
  | charConst
  | "new" ident [ "[" Expr "]" ]
  | "(" Expr ")"
  | "(" Type ")" Expr.
  ```
  The new alternative enables type casts. It creates an LL(1) conflict, since the alternative `"(" Expr ")"` also starts with a left parenthesis.

The implementation of these changes and the solutions to the LL(1) conflicts are presented in the next subsections. The complete MicroJava language specification can be found in Appendix A.

3.1.1. Inheritance

In preparation for the implementation of inheritance, we changed some of the nodes of the symbol table. `Struct` nodes received the following new fields:

- `superType`, points to the node of the superclass.
- `level`, contains the level (or depth) of the class in the class hierarchy.
- `nMethods`, contains the number of methods declared in the respective class and its superclasses, excluding methods that are overwritten.
- `tdAdr`, holds the type descriptor address of the type. The type descriptor contains information about the methods and superclasses of a class.
- `locals`, references the fields and methods of the class, renamed from `fields`.

Methods can now also be declared in classes. Methods declared in classes are stored in the `locals` of a `Struct` node, just as fields are. Every method `Obj` node needs a method number that is unique for all methods in the `locals` of a `Struct` node. However, an overwriting method gets the same method number as the method it overwrites. The
  ```

method number is used to resolve the method that is called based on the dynamic type of the object during run-time, since the dynamic type is not known during compile-time.

The productions `VarDecl` and `MethodDecl` can both start with an identifier. The parser therefore cannot decide if it is parsing a variable declaration or a method declaration in a class declaration when it reads an identifier. This LL(1) conflict is solved by checking if the current look-ahead token is a `void` token, since a variable cannot have the type void, or by looking ahead for a left parenthesis, which marks a method declaration.

```
class Parser {
  ...
  private void ClassDecl() {
    ...
    while (sym == ident || sym == void_) {
      if (isMethodDecl()) {
        MethodDecl();
      } else {
        VarDecl();
      }
    }
    ...
  }
  private boolean isMethodDecl() {
    return sym == void_ || scanner.peek(2).kind == lpar;
  }
}
```

Listing 2: Solution of the LL(1) conflict in ClassDecl

When a new `Struct` node is created for a subclass, the node is initialized with values based on the `superType`. Similarly, the `Scope` keeps track of the number of fields and methods when a class is parsed. Since fields and methods may be declared in superclasses, the `findField()` method and the new `findMethod()` method have to follow the `superType` pointers until the searched object is found or no superclass exists.

Within class methods, fields and methods can be accessed either through `this` or without a qualifier. Therefore, the methods that find objects have to account for the different kinds of qualified accesses. In particular, the `find()` method may now also have to search the class hierarchy to find an object.

Objects of a subclass are assignable to variables of a superclass. This property has to be checked when parsing assignments by searching for the destination type in the class hierarchy of the source type, which is shown in Listing 3.

```
public final class Struct {
    ...
    public boolean assignableTo(Struct dest) {
        return this.isEqual(dest)
                || this == Tab.nullType && dest.isRefType()
                || this.kind == Kind.Arr && dest.kind == Kind.Arr
                    && dest.elemType == Tab.noType
                || this.kind == Kind.Class && dest.kind == Kind.Class
                    && this.isSubtypeOf(dest);
    }
    public boolean isSubtypeOf(Struct other) {
        Struct type = this.superType;
        while (type != null && type != other) {
            type = type.superType;
        }
        return type == other;
    }
}
```

Listing 3: Checking the assignability of two types

Once the compiler has parsed the entire input, information about the declared classes is written into the object file. This includes the number of classes and, for each class, the number of methods and the method addresses. The loader reads this information, builds a table in the heap containing the method addresses of a class, and initializes the entries in the global data with the addresses of the tables. Such a table resembles a type descriptor.

Every object allocated during run-time contains a hidden field called type tag that points to the type descriptor of its dynamic type. When a new object is allocated, the interpreter reads the address of the type descriptor address, which is given by the byte code, looks up the type descriptor address in the global data, and initializes the type tag of the allocated object.

When the interpreter wants to call a method of an object, it performs a virtual call. Given the address of the object and the method number, the interpreter looks up the address of the method in the type descriptor of the dynamic type of the object.

Listing 4 and Figure 3.1 show how inheritance is used in MicroJava code and how it is represented in the symbol table.

```
1  class A {
2      int a;
3      int b;
4      void M (int x) { ... }
5      int N () { ... }
6  }
7  class B extends A {
8      int c;
9      int N () { ... }
10     void S () { ... }
11 }
```

Listing 4: MicroJava class declarations

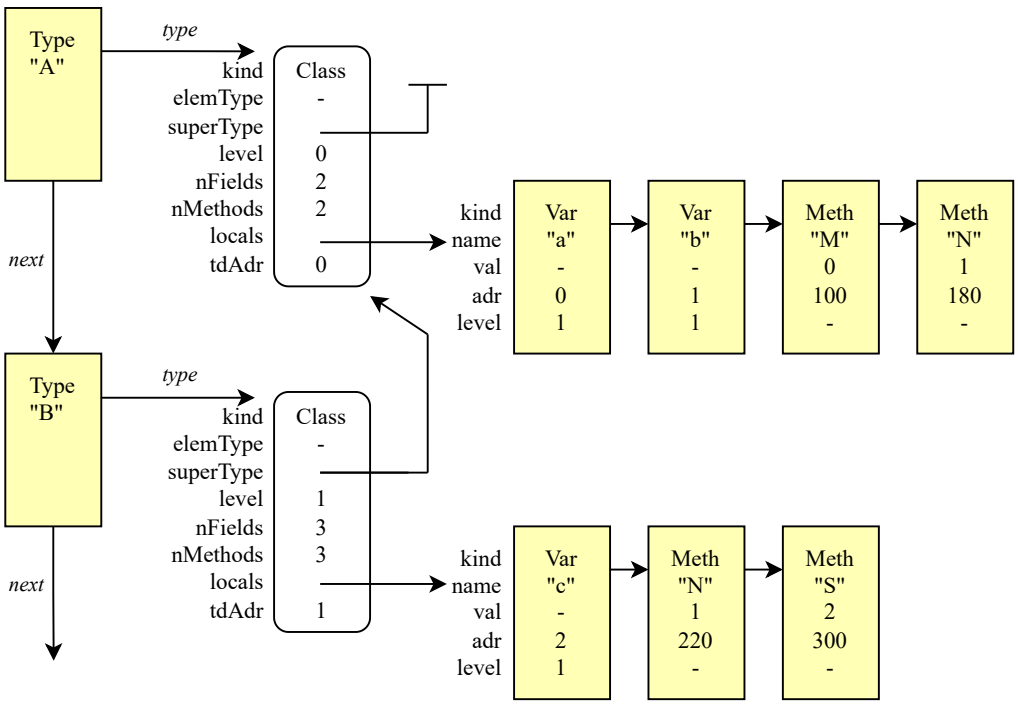


Figure 3.1.: Symbol table nodes created by the class declarations in Listing 4. Yellow nodes are `Obj` nodes, white nodes are `Struct` nodes.

3.1.2. Type check and type cast

Type checks with the `instanceof` operator check if an object is of the same class as a given class or a subclass. Checking for the same class is easy, but checking for a subclass during run-time requires more information to be stored. To accomplish this, the compiler writes the type descriptor addresses of all superclassses of each class in the object file. The loader reads the addresses and initializes a base type table in the heap for each class, which is part of the type descriptor.

```
ObjFile   = "MJ" codeSize dataSize mainPC
            nClasses { nLevels { tdAdr } nMethods { methodAdr } }
            Code.
Code      = { byte }.
```

Listing 5: (Preliminary) object file format

For simplicity, the base type table has a static size of 4 entries. This restriction simplifies the usage of the type descriptor. For example, the method table can be accessed with a static offset of 4 words. If a class hierarchy has fewer than 4 classes, the remaining entries in the table are empty.

Using the base type table, type checks are very efficient. Given the example `if (obj instanceof B)`, assuming the declarations from Listing 4, the interpreter accesses the type descriptor of `obj` through the type tag and checks if the type descriptor address at the level of the type `B` is actually the type descriptor address of `B`. If this is the case, the interpreter pushes '1' on the expression stack, else '0'. This is how the new instruction `checkcast` works.

Type checks in conditions can use the result of the `checkcast` instruction to evaluate the condition and jump to the right position using the new instructions `jt` (jump on true) and its complement `jf` (jump on false).

Type casts use the `checkcast` instruction too. The difference to type checks is that if `checkcast` returns 0, the cast fails and the program execution stops. Throwing a MicroJava exception (e.g., a `ClassCastException` as it would happen in Java) that can be caught within MicroJava is not possible, since there are no predefined classes for exceptions.

Type casts cause an LL(1) conflict in the MicroJava grammar because two alternatives in the production Factor start with a left parenthesis. However, the conflict can be resolved

by reading the left parenthesis and then deciding which alternative is parsed based on the next token. Both following non-terminal symbols, Expr and Type, can start with an identifier, which again is an LL(1) conflict. This time, the conflict is easily resolved by checking the kind of the identifier, as only the production Type can start with an identifier that denotes a type.

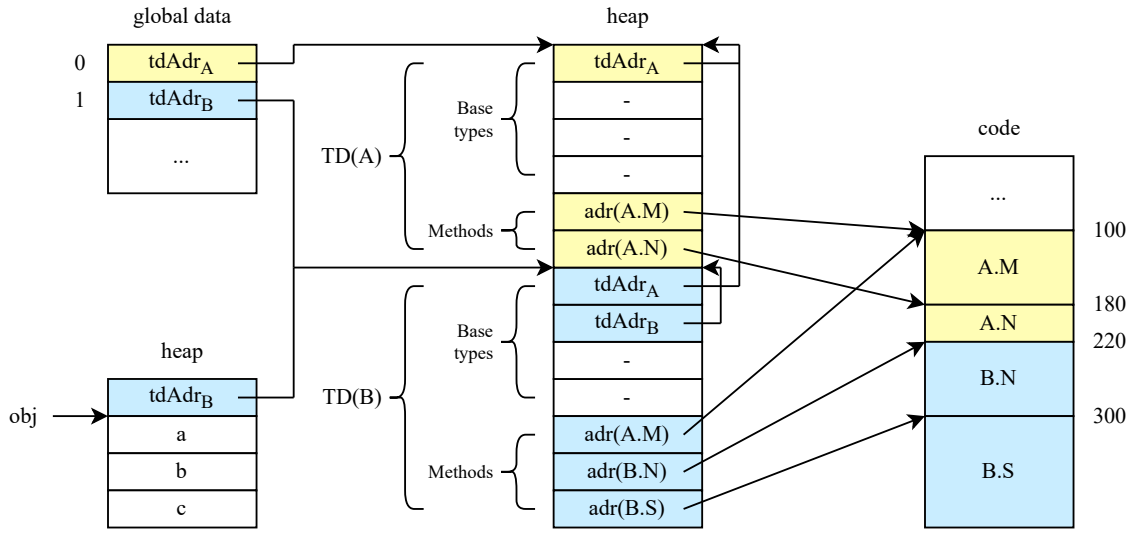


Figure 3.2.: Potential memory layout given the declarations in Listing 4 and the statement `B obj = new B;`

3.2. Exception handling

Exception handling requires three new keywords: `try`, `catch` and `throw`. The MicroJava grammar only received small changes. Two new alternatives were added to the production Statement:

```
Statement = ...
| "try" Block "catch" "(" ident ")" Block { "catch" "(" ident ")" Block }
| "throw" Expr ";".
```

The first alternative enables `try-catch` clauses. A `try` block must have at least one `catch` block, as is the case in Java. However, unlike in Java, it is not possible to declare local

variables within a method block. Due to this restriction, `catch` parameters have to be declared beforehand as local variables of the function. It is theoretically also possible to use global variables as catch parameters, but that is very unusual.

The second alternative allows an object to be thrown as an exception. The type of the object must be a class, but since there is no predefined exception class or interface that has to be extended or implemented like in Java, objects of any class can be thrown.

In MicroJava exception handling works as follows: After an exception is thrown, the exception handler looks for a `catch` parameter in `try-catch` blocks that the exception object can be assigned to. An exception can be assigned to a `catch` parameter if the exception is of the same class as the parameter or a subclass. If there is no `try-catch` clause or no fitting `catch` parameter in the current method frame, the frame is exited by restoring stack pointer and frame pointer, and the search for a catch parameter is started again at the return address. If the main method is exited without the exception being caught, the program execution is stopped. If the exception handler finds a fitting `catch` parameter, the exception object is assigned to the parameter and the respective `catch` block is executed.

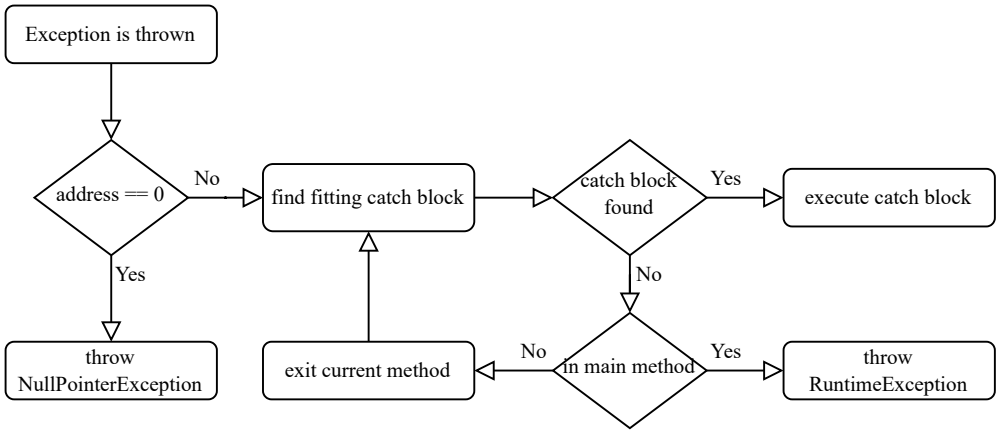


Figure 3.3.: Flow chart of the exception handling process

To be able to find a catch parameter, the exception handler needs to have information about all existing `try-catch` clauses. This is accomplished by collecting information about `try-catch` clauses during the compilation. For each `catch` block, the compiler writes the start and the end address of the respective `try` block, the type descriptor address of the

`catch` parameter and the start address of the `catch` block to the object file. The loader then creates an exception table that is used by the exception handler.

```
ObjFile   = "MJ" codeSize dataSize mainPC
            nClasses { nLevels { tdAdr } nMethods { methodAdr } }
            nCatches { fromAdr toAdr tdAdr_Exc catchAdr_Exc }
            Code.
Code      = { byte }.
```

Listing 6: Final object file format

The expression stack requires special handling during exception handling. Looking at the example statement `a[i] = foo();`, the caller puts the address of the array `a` and the value of `i` onto the expression stack in preparation of the assignment. If the method `foo()` returns normally, these values should still be on the expression stack after `foo()` returned. However, if an exception occurs, the exception handler exits the method and searches for a `catch` clause in the frame of the callee, which means that the assignment is now irrelevant. Yet, the address of `a` and the value of `i` are still on the expression stack. Because of this, the exception handler has to manually restore the expression stack to the state it was in before the statement. Additionally, since the `throw` statement pushes the exception object onto the expression stack, the object has to be removed at the start of the exception handling and pushed onto the stack again at the end.

To be able to restore the expression stack when exiting a method frame, the expression stack pointer has to be pushed onto the method stack when a method is entered. This happens in the `enter` instruction, after the frame pointer is pushed. When a method is exited normally, the `exit` instruction removes the expression stack pointer again.

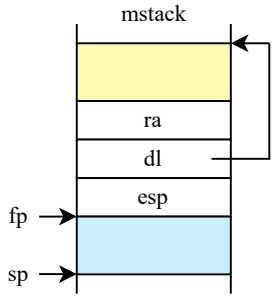


Figure 3.4.: Layout of the method stack

```
public class Interpreter {
  ...
  public void run() {
    ...
    switch (op) {
      ...
      case throw_ -> {
        adr = pop();
        if (adr == 0) {
          throw new NullPointerException("null reference used");
        }
        int throwAdr = pc;
        while (true) {
          val = getHandler(throwAdr, adr); // search for catch clause
          if (val >= 0) {
              push(adr); pc = val; break;
          }
          sp = fp;
          POP(); // remove old esp
          fp = POP();
          if (fp == 0) { // if we try to exit main -> stop
            throw new RuntimeException("uncaught exception");
          }
          esp = local[fp - 1]; // restore esp
          throwAdr = POP();
        }
      }
    }
    ...
  }
}
```

Listing 7: Exception handling in the interpreter

4. Tests

This section describes the tests that ensure the correctness of the implementation of the compiler. The new tests are split into five test classes: `ObjectOrientationTest`, `AssingmentCheckcastTest`, `DynamicBindingTest`, `ExceptionHandlingTest` and `CatchTypeTest`. The first three classes contain tests for the object orientation features, the latter two for exception handling.

All test classes use the utility methods provided by the `CompilerTestCaseSupport` class. The most important method is `parseAndVerify()`, which parses the input and checks if expected errors occurred, the symbol table looks as expected, or test runs produce a certain output. We extended the functionality of the method to also allow for checking for thrown exceptions and their messages.

We used the JUnit 4 testing framework in this project.[1] The tests are divided into two categories: standalone tests and systematic tests. Standalone tests are normal unit tests that test different kinds of aspects of the system. `ObjectOrientationTest` and `ExceptionHandlingTest` contain standalone tests.

Systematic tests test one specific aspect with multiple configuration features, where each combination of configurations is tested systematically. The systematic tests are implemented using the `Parameterized` custom test runner, which allows us to run a test with multiple predefined configurations. `AssignmentCheckcastTest`, `DynamicBindingTest` and `CatchTypeTest` contain systematic tests.

[1] `https://junit.org/junit4/`

4.1. Object orientation

Three test classes test the object orientation features: `ObjectOrienationTest`, `Assignment-CheckcastTest` and `DynamicBindingTest`.

`ObjectOrientationTest` contains standalone tests that test the basic functionality of the object orientation features. The tests check if the compiler can parse class declarations, class method declarations, type checks, and type casts, if it correctly represents inheritance in the symbol table, and whether the interpreter implements the functionality of the features correctly in test runs.

4.1.1. Assignments, type checks and type casts

`AssignmentCheckcastTest` takes a closer look at the assignability of objects to variables depending on their dynamic and static types, respectively, as well as type checks and type casts. The class features three tests: `testAssignment()`, `testTypeCheck()` and `testTypeCast()`. All three tests of the class `AssignmentCheckcastTest` use the class declarations in Listing 8.

```
1 class X {}
2 class Y extends X {}
3 class Z extends Y {}
```

Listing 8: Class declarations used in `AssignmentCheckcastTest`

All tests use the types T1 and T2 as parameters. The tests are executed for all elements (T1, T2) of the cross product of {X, Y, Z} and {X, Y, Z}.

The test `testAssignment()` checks if the assignment of an object of the dynamic type T2 to a variable of the static type T1 in Listing 9 parses and runs without an error.

The test `testTypeCheck()` tests if the type check in Listing 10 evaluates to true, which happens if the type T2 is equal to or a subtype of the type T1.

The last test, `testTypeCast()`, checks if the code in Listing 11 successfully casts an object of type T2 to the type T1 without throwing a `ClassCastException`.

The expected results of the three tests are shown in Table 4.1. The descriptions of the tests above describe the 'positive' case, which is marked with a checkmark (✓) in the result tables. The 'negative' outcome is marked with a cross (✗).

```
void main() T1 t; {
  t = new T2;
}
```

Listing 9: MicroJava code tested in `testAssignment()`

```
void main() X t; {
  t = new T2;
  if (t instanceof T1) print(1);
}
```

Listing 10: MicroJava code tested in `testTypeCheck()`

```
void main() X t; {
  t = (T1) new T2;
}
```

Listing 11: MicroJava code tested in `testTypeCast()`

		T2		
		X	Y	Z
T1	X	✓	✓	✓
	Y	✗	✓	✓
	Z	✗	✗	✓

Table 4.1.: Expected results of `testAssign()`, `testTypeCheck()` and `testTypeCast()`

The reader will notice that the results of all three tests are the same. This is due to the fact that the 'source' type of assignments, type checks, and type casts has to be equal to or a subclass of the 'target' type. This is the reason all three features are tested in one test class.

4.1.2. Dynamically bound method calls

The test class `DynamicBindingTest` tests the execution of dynamically bound method calls, or virtual method calls, by trying to call a method `foo()` on an object. The MicroJava code that is executed is shown in Listing 12. There are three parameters to this test:

- Where is the method `foo()` declared?
- What is the static type T1 of t?
- What is the dynamic type T2 of t?

```
1  program Test
2  class X { [ void foo() { print('X'); } ] }
3  class Y extends X { [ void foo() { print('Y'); } ] }
4  class Z extends Y { [ void foo() { print('Z'); } ] }
5  {
6    void main() T1 t; {
7      t = new T2;
8      t.foo();
9    }
10 }
```

Listing 12: MicroJava code tested in `DynamicBindingTest`

The three parameters give us $2^3 \cdot 3 \cdot 3 = 72$ combinations of parameters. However, it does not make sense to test all of these combinations. Cases where the dynamic type is not assignable to the static type were left out, as this is already tested in `AssignmentCheckcastTest` in Section 4.1.1. This leaves us with 48 combinations.

There are two types of events that can happen: Either the method is not accessible, which results in a compilation error, or it is accessible and the name of the class the executed method is declared in is printed to the console. The results of the tests are shown in Table 4.2. An entry in the table contains either the name of the class the executed method was declared in or a dash (-) if the method is not accessible.

Stat.	Dyn.	`foo()` defined in							
		none	X	Y	Z	X, Y	X, Z	Y, Z	X, Y, Z
X	X	-	X	-	-	X	X	-	X
	Y	-	X	-	-	Y	X	-	Y
	Z	-	X	-	-	Y	Z	-	Z
Y	Y	-	X	Y	-	Y	X	Y	Y
	Z	-	X	Y	-	Y	Z	Z	Z
Z	Z	-	X	Y	Z	Y	Z	Z	Z

Table 4.2.: Expected results of `DynamicBindingTest`

4.2. Exception handling

Just like the object orientation feature, exception handling is tested by standalone tests in `ExceptionHandlingTest` and systematic tests in `CatchTypeTest`. Standalone tests test all aspects of the feature in separate tests, including the correct parsing and execution of exception handling.

The test class `CatchTypeTest` checks which catch block is executed depending (1) on the type T of the thrown exception and (2) on which catch blocks are defined. The structure of the MicroJava code that is executed in the test is shown in Listing 13. The catch blocks are sorted by the type of the catch parameter, starting with the most specific (sub)class and continuing with the more general (super)classes. Otherwise, the exception would always be caught by the first, more general catch block.

The expected results of the test are shown in Table 4.3. If the exception was caught, the entry in the table contains the type of the catch parameter. Otherwise, the exception was not caught, which is marked with a dash (-).

```
program Test
class X {}
class Y extends X {}
class Z extends Y {}
{
  void main() X x; Y y; Z z; {
    try {
      throw new T;
    } [catch (z) {
      print('z');
    } ][catch (y) {
      print('y');
    } ][catch (x) {
      print('x');
    } ]
  }
}
```

Listing 13: MicroJava code tested in `CatchTypeTest`

Exc.	**defined `catch` blocks**						
	X	Y	Z	X, Y	X, Z	Y, Z	X, Y, Z
X	X	-	-	X	X	-	X
Y	X	Y	-	Y	X	Y	Y
Z	X	Y	Z	Y	Z	Z	Z

Table 4.3.: Expected results of `CatchTypeTest`

5. Conclusion and future work

In this project, we successfully implemented the basic features of object orientation and exception handling. We added inheritance, class methods, virtual method calls, type checks, and type casts, as well as `try-catch` blocks and `throw` statements to the MicroJava language. We changed the compiler to be able to parse the new features and added new instructions that the interpreter executes. Finally, we added unit tests that ensure the correctness of the implementation of the new features.

However, some features we could only implement with restrictions. For example, we did not add the `super` identifier additionally to `this` due to difficulties when checking the formal and actual parameters of a class method. Overwriting fields or giving a field and a class method the same name is not possible in the current version of MicroJava. The syntax of `catch` blocks differs from Java, as it is not possible to declare variables inside a method block.

Future work could try to remove these restrictions by further expanding the capabilities of the MicroJava language. Furthermore, there are many features of Java that could be implemented next, such as access modifiers, non-primitive method return types, or method overloading.

Appendices

A. MicroJava specification

This section contains the specification of the MicroJava language and virtual machine, including all changes made in this project. The original specification was provided by the lecturers of the compiler construction course at the Institute of System Software.

A.1. Die Sprache MicroJava

Dieser Abschnitt beschreibt die in den [Anm.: Compilerbau-] Übungen verwendete Sprache MicroJava. Sie ist an Java angelehnt, jedoch wesentlich vereinfacht.

A.1.1. Allgemeine Merkmale

- Ein MicroJava-Programm besteht aus einem *program*-Konstrukt mit statischen Daten und statischen Methoden. Das gesamte Programm steht in einer einzigen Datei.
- Die Hauptmethode heißt *main()*. Bei ihr beginnt die Programmausführung.
- Es gibt
  - Konstanten: *int*-Konstanten (z.B. 3) und *char*-Konstanten (z.B. 'x'), aber keine Stringkonstanten.
  - Variablen: die Variablen des Hauptprogramms sind statisch.
  - Werttypen: *int*, *char* (Ascii)
  - Referenztypen: eindimensionale Arrays wie in Java sowie Klassen mit Feldern und Methoden. Variablen dieser Typen enthalten Referenzen.

  - Statische Methoden des Hauptprogramms.

- Es gibt keinen Garbage-Collector (angelegte Objekte bleiben einfach übrig).
- Standardprozeduren sind *ord*, *chr* und *len*.

A.1.2. Syntax

```
Program    = "program" ident {ConstDecl | VarDecl | ClassDecl}
             "{" {MethodDecl} "}".
ConstDecl  = "final" Type ident "=" (number | charConst) ";".
VarDecl    = Type ident {"," ident} ";".
ClassDecl  = "class" ident ["extends" Type] "{" {VarDecl | MethodDecl} "}".
MethodDecl = (Type | "void") ident "(" [ ormPars] ")" {VarDecl} Block.
FormPars   = Type ident {"," Type ident}.
Type       = ident ["[" "]"].
Block      = "{" {Statement} "}".
Statement  = Designator (Assignop Expr | "++" | "--" | ActPars) ";"
           | "if" "(" (Condition) ")" Statement ["else" Statement]
           | "while" "(" (Condition) ")" Statement
           | "break" ";"
           | "return" [Expr] ";"
           | "read" "(" Designator ")" ";"
           | "print" "(" Expr ["," number] ")" ";"
           | Block
           | ";"
           | "try" Block "catch" "(" ident ")" Block
             {"catch" "(" ident ")" Block}
           | "throw" Expr ";".
Assignop   = "=" | "+=" | "-=" | "*=" | "/=" | "%=".
ActPars    = "(" [Expr {"," Expr}] ")".
Condition  = CondTerm {"||" CondTerm}.
CondTerm   = CondFact {"&&" CondFact}.
CondFact   = Expr (Relop Expr | "instanceof" Type).
Relop      = "==" | "!=" | ">" | ">=" | "<" | "<=".
Expr       = ["-"] Term {Addop Term}.
Term       = Factor {Mulop Factor}.
```

```
Factor     = Designator [ActPars]
           | number
           | charConst
           | "new" ident ["[" Expr "]"]
           | "(" Expr ")"
           | "(" Type ")" Expr.
Designator = ident {"." ident | "[" Expr "]"}.
Addop      = "+" | "-".
Mulop      = "*" | "/" | "%".
```

Lexikalische Struktur

Zeichenklassen:

```
letter     = 'a' .. 'z' | 'A' .. 'Z'.
digit      = '0' .. '9'.
whiteSpace = ' ' | '\t' | '\r' | '\n'.
```

Terminalklassen:

```
ident      = letter {letter | digit | '_'}.
number     = digit {digit}.
charConst  = "'" char "'".
             // einschließlich '\r', '\n', '\'' und '\\'
```

Schlüsselwörter:

```
program class
if else while read print return break void final new
extends instanceof try catch throw
```

Operatoren:

```
+    -    *    /    %    ++   -
==   !=   >    >=   <    <=
&&   ||
(    )    [    ]    {    }
=    +=   -=   *=   /=   %=
;    ,    .
/* ...  */ können auch geschachtelt werden
```

A.1.3. Semantik

Alle in diesem und im nächsten Abschnitt vorkommenden Begriffe, zu denen es eine Definition gibt, sind unterstrichen, um ihre besondere Bedeutung auszudrücken.

Referenztypen

Arrays und Klassen sind Referenztypen.

Typen von Konstanten

- Der Typ einer Zahlkonstante ist *int*.
- Der Typ einer Zeichenkonstante ist *char*.

Typgleichheit

Zwei Typen sind gleich,

- wenn sie durch denselben Typnamen ausgedrückt sind, oder
- wenn beide Typen Arrays und ihre Elementtypen gleich sind.

Typkompatibilität

Zwei Typen sind kompatibel

- wenn sie gleich sind, oder
- wenn einer ein Referenztyp ist und der andere Parameter den Wert null hat.

Zuweisungskompatibilität

Ein Typ *src* ist mit einem Typ *dst* zuweisungskompatibel

- wenn *src* und *dst* gleich sind, oder
- wenn *dst* ein Subtyp von *src* ist, oder
- wenn *dst* ein Referenztyp ist und *src* der Typ von null.

Vordeklarierte Namen

int	Standardtyp
char	Standardtyp
null	Nullwert einer Klassen- oder Arrayvariable
this	Objekt, dessen Methode gerade ausgeführt wird
chr	Standardmethode; *chr(i)* wandelt den int-Ausdruck *i* in einen *char*-Wert um
ord	Standardmethode; *ord(ch)* wandelt den *char*-Wert *ch* in einen *int*-Wert um
len	Standardmethode; *len(a)* liefert die Anzahl der Elemente des Arrays *a*

Scope

Ein Scope ist der textuelle Bereich einer Methode oder Klasse. Er erstreckt sich vom Punkt nach dem Methoden- oder Klassennamen bis zur schließenden geschweiften Klammer. Ein Scope schließt die in ihn eingeschachtelten Scopes aus. Wir nehmen an, dass es einen (künstlichen) äußersten Scope gibt, in den die Hauptklasse eingeschachtelt ist, und in dem alle vordeklarierten Namen deklariert sind (= "Universum"). Die Deklaration eines Namens in einem inneren Scope *S* überdeckt in *S* eine eventuelle Deklaration des gleichen Namens in einem äußeren Scope.

Hinweise

- Indirekte Rekursion ist nicht erlaubt, da jeder Name vor seiner Verwendung deklariert werden muss, was bei indirekter Rekursion nicht möglich ist.
- Ein vordeklarierter Name (z.B. *int*, *char*) kann durch eine neuerliche Deklaration in einem inneren Scope überschrieben werden (was aber nicht zu empfehlen ist).

A.1.4. Kontextbedingungen

Allgemeine Kontextbedingungen

- Jeder Name muss vor seiner Verwendung deklariert worden sein.
- Kein Name darf im gleichen Scope mehr als einmal deklariert sein.
- Das Programm muss eine Methode namens "main" enthalten. Diese muss als void deklariert sein und darf keine Parameter haben.

Kontextbedingungen für Standardmethoden

chr(e) *e* muss ein Ausdruck vom Typ *int* sein.
ord(c) *c* muss vom Typ *char* sein.
len(a) *a* muss ein Array sein.

Kontextbedingungen für die einzelnen Grammatikregeln

Nur abgeänderte Regeln sind hier aufgelistet.

ClassDecl = "class" ident ["extends" Type] "{" {VarDecl | MethodDecl} "}".

- *Type* muss eine Klasse sein.
- Wenn eine Methode eine andere überschreibt, müssen beide Methoden denselben Typ und dieselben Parameter haben.

Statement = "try" Block "catch" "(" ident ")" Block {"catch" "(" ident ")" Block}.

- *ident* muss eine lokale oder globale Variable bezeichnen.
- Der Typ von *ident* muss eine Klasse sein.

Statement = "throw" Expr ";".

- Der Typ von *Expr* muss eine Klasse sein.

CondFact = Expr "instanceof" Type.

- Der Typ von *Expr* und *Type* müssen beide Klassen sein.
- Der Typ von *Expr* und *Type* müssen sich beide in der selben Klassenhierarchie befinden.

Factor = "(" Type ")" Expr.

- Der Typ von *Expr* und *Type* müssen beide Klassen sein.
- Der Typ von *Expr* und *Type* müssen sich beide in der selben Klassenhierarchie befinden.

Designator_0 = Designator_1 "." ident.

- Der Typ von $Designator_1$ muss eine Klasse sein.
- *ident* muss ein Feld oder eine Methode von $Designator_1$ sein.

A.1.5. Implementierungsbeschränkungen

- Es darf nicht mehr als 127 lokale Variablen geben.
- Es darf nicht mehr als 32767 globale Variablen und Klassendeklarationen geben.
- Eine Klasse darf nicht mehr als 32767 Felder und 127 Methoden haben.
- Eine Klasse darf nicht mehr als 3 Supertypen haben.

A.2. Die MicroJava VM

Dieses Kapitel beschreibt die Architektur der in den [Anm.: Compilerbau-] Übungen verwendeten virtuellen Java-Maschine. Sie ist an die Architektur der echten Java VM angelehnt, enthält aber weniger Befehle. Einige Befehle wurden vereinfacht. Wo die Java VM als Operanden Einträge des Konstantenpools verwendet, benutzt die MicroJava VM fixe Operanden. Das erspart die Einführung eines Konstantenpools. Ferner ist in MicroJava-Befehlen nicht der Typ der Operanden codiert, wie das bei der Java VM der Fall ist.

Dieses Kapitel enthält nur die Änderungen, die während dieses Projekts vorgenommen wurden.

A.2.1. Speicher-Layout

Ein neuer Speicherbereich wurde hinzugefügt: die Laufzeit-Exception-Tabelle. Der Speicher ist ein zweidimensionales Wortarray. Hier liegen die Einträge der catch-Klauseln eines `try-catch`-Konstrukts. Ein Eintrag enthält die Adresse des Anfangs des `try`-Blocks im Code, die Adresse des Endes des `try`-Blocks, die Adresse des Anfangs des `catch`-Blocks sowie die Typdeskriptoradresse der Variable, der die geworfene Exception zugewiesen wird.

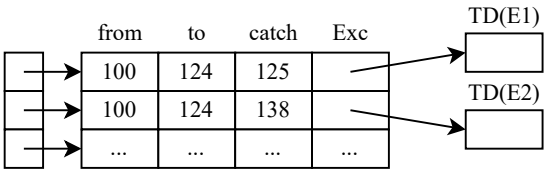


Figure A.1.: Layout der Laufzeit-Exception-Tabelle

Außerdem wurde für die Implementierung der Ausnahmebehandlung die Struktur eines Aktivierungssatzes (Frame) im Methodenstack angepasst. Ein Frame enthält nun neben der Rücksprungadresse *ra* und dem Dynamic Link *dl* den *ExprStack* Pointer *esp*.

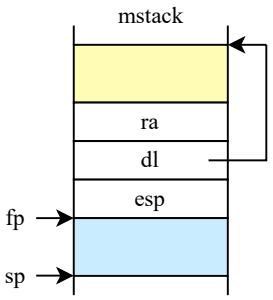


Figure A.2.: Layout des Methodenstacks

A.2.2. Befehlssatz

Die folgenden Tabellen zeigen die Befehle der MicroJava-VM zusammen mit ihrem Befehlscode und ihrer Wirkungsweise. Die dritte Spalte der Tabellen zeigt den Inhalt von ExprStack vor und nach dem jeweiligen Befehl an, z.B. bedeutet

```
..., val, val
..., val
```

dass der Befehl zwei Worte vom ExprStack entfernt und ein neues Wort auf ExprStack legt. Die Operanden des Befehls haben folgende Bedeutung:

b ein Byte (8 Bit, vorzeichenbehaftet)
s ein Shortint (16 Bit, vorzeichenbehaftet)
w ein Wort (32 Bit, vorzeichenbehaftet)

Variablen vom Typ char werden im niederwertigsten Byte eines Worts gespeichert und mit Wort-Befehlen manipuliert (z.B. load, j<cond>). Arrayelemente vom Typ char werden in einem Bytearray gehalten und mit speziellen Befehlen geladen und gespeichert.

Die Tabelle beinhaltet nur veränderte oder neu hinzugefügte Befehle. Die erste Spalte beinhaltet die Nummer des veränderten Befehls, oder "neu", falls der Befehl neu ist.

32	**new** s1 s2	...	New object
		..., adr	n = s1; tdAdr = data[s2];
			alloc area of n *words*;
			initialize area to 0;
			area[0] = tdAdr;
			push(adr(area) + 1);
neu	**checkcast** s b	..., obj	Dymanic type test
		..., 0 | 1	tdAdr = data[s]; level = b;
			obj = pop();
			tag = heap[obj - 1];
			push(tag.types[level] == tdAdr ? 1 : 0);
neu	**jt** s	..., n	Jump on true
		...	if (pop() == 1) pc = pc + s;

neu	**jf** s	..., n	Jump on false
		...	if (pop() == 0) pc = pc + s;
neu	**vcall** b1 b2	..., parameters	Virtual call of a method
		..., parameters	m = b1; nPars = b2;
			obj = estack[esp - nPars];
			tab = heap[obj - 1];
			PUSH(pc + 3);
			pc = tab[m];
51	**enter** b1 b2	...	Enter method
		...	nPars = b1; nVars = b2;
			PUSH(fp); PUSH(esp);
			fp = sp; sp = sp + nVars;
			initialize frame to 0;
			for (i = nPars - 1; i >= 0; i–)
			local[i] = pop();
52	**exit**	...	Exit method
		...	sp = fp; POP(); fp = POP();
neu	**throw**	..., e	Throw exception
		..., e	e = pop();
			throwAdr = pc;
			while (true) {
			h = getHandler(throwAdr, e);
			if (h >= 0) { pc = h; push(e); break; }
			sp = fp;
			POP();
			fp = POP();
			if (fp == null) { *error*; *stop execution*; }
			esp = locals[fp - 1];
			throwAdr = POP();
			}

Table A.1.: Befehle der MicroJava VM

A.2.3. Objektdateiformat

ObjFile = "MJ" codeSize dataSize mainPC
nClasses { nLevels { tdAdr } nMethods { methodAdr } }
nCatches { fromAdr toAdr $tdAdr_{Exc}$ $catchAdr_{Exc}$ }
Code.
Code = { byte }.

A.2.4. Laufzeitfehler

1. Missing return statment in function.
2. Incompatible types for type casting.

Bibliography

[1] P. M. Lewis and R. E. Stearns. "Syntax-Directed Transduction". In: *J. ACM* 15.3 (July 1968), pp. 465–488. ISSN: 0004-5411. DOI: 10.1145/321466.321477. URL: https://doi.org/10.1145/321466.321477 (cit. on p. 3).

[2] B. Stroustrup. "What is object-oriented programming?" In: *IEEE Software* 5.3 (May 1988), pp. 10–20. DOI: 10.1109/52.2020 (cit. on p. 3).

[3] John B. Goodenough. "Exception Handling: Issues and a Proposed Notation". In: *Commun. ACM* 18.12 (Dec. 1975), pp. 683–696. ISSN: 0001-0782. DOI: 10.1145/361227.361230. URL: https://doi.org/10.1145/361227.361230 (cit. on p. 5).